Zaluski, A. (2001). Knowledge Spaces Mathematica Package (Unpublished Technical Report). Institut für Psychologie Karl–Franzens–Universität Graz, Austria.

Knowledge Spaces *Mathematica* Package

Technical Report

Andrej Zaluski [†]

University of Graz

Version 1.0
May, 2001.

[†]Address: Andrej Zaluski, Institut für Psychologie, Abteilung für Allgemeine Psychologie, Karl-Franzens-Universität Graz, Universitätsplatz 2/3, A-8010 Graz, Austria. E-mail: andrej.zaluski@uni-graz.at

Acknowledgements

The work on *Knowledge Spaces Mathematica Package* was supported by the Ernst-Mach fellowship (BMBWK G-1621/*Querying Experts on Skills*) to the author. The author would like to thank Dietrich Albert, Cord Hockemeyer and Christof Körner for support during the research.

License

1. This software package is provided "as is", but in hope to be useful. No warranty either explicit or implicit is given, no claim on fitness to any particular purpose made. The author does not accept responsibility for consequences of using the package. The entire risk as to the quality and performance of the program is with you. The permission to run the package with no restrictions is given under these terms.

2. Permission is granted to anyone to make or distribute verbatim copies of the package provided that the copyright notice and the permission notice are preserved, and that the distributor grants the recipient permission for further redistribution as permitted by this notice.

3. You may modify your copy or copies of the package or any portion of it, thus forming a work based on the package, and copy and distribute such modifications or work under the terms of section 2 provided that you also meet the following conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in parts contains or is derived from the package or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

You may not run, copy, modify, sublicense, or distribute the package except as expressly provided under this license. Any attempt otherwise to run, copy, modify, sublicense or distribute the package is void, and will automatically terminate your rights under this License. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to run, modify or distribute the package or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by running, distributing or modifying the package (or any work based on the package), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the package or works based on it.

Overview

This paper describes Knowledge Spaces *Mathematica* Package version 1.0. Firsty, motivation for developing the package is presented and *Mathematica* is described as a platform suitable for research and teaching within the field of knowledge spaces. The software and hardware requirements are presented together with portability issues and instructions for installing the package. The forms of package documentation are briefly described, and some hints on using the on-line documentation given. The package design is presented by describing the package architecture and used file formats. The package is conceived as consisting of four abstract layers. In this package version, the first two layers, together with the warp core module, forms the core of the package. The layer three is comprised of different modules, i.e. skills module, querying module and assessment module. Used data file formats are described in the sequel. A plan to equip the fourth layer with Querying Experts on Skills application, together with further development of the core, enhancements and extensions of the modules are briefly presented. The alphabetical glossary containing all public package functions makes the central part of the paper. Finally, a bibliography and the source code of the package are provided.

Contents

Acknowledgements...2
License...3
Overview...4
Contents...5
Introduction...6
    Motivation and development of the package...6
    Mathematica as a platform suitable for research and teaching within the field of knowledge spaces...6
Installation...7
    Software and hardware requirements;portability and versioning...7
    Installing the package...7
    Package documentation...8
Package design...9
    Architecture of the package...9
        KSMP Layer one...9
        KSMP Warp core...10
        KSMP Layer two...10
        KSMP Skills module...10
        KSMP Querying module...11
        KSMP Assessment module...11
    File formats used...12
        Spacefile format...12
        Itemsfile format...12
        Skillsfile format...13
Further developments...14
Bibliography...15
Glossary of Functions...16
Source code...42

Introduction

Motivation and development of the package

Historically, *Knowledge Spaces Mathematica Package* (*KSMP*) started as a part of *Querying Experts on Skills* (*QEOS*) project. The selection of knowledge space theory concepts currently implemented in the package may be seen as a simple consequence of this fact. The initial task within the project was to find a suitable software platform that would facilitate research and development of querying procedures that besides observable problem-solving behavior also take into account latent skills and competencies. Such a platform should primarily be suitable for theoretical explorations and simulation studies. The platform should also be suitable for applications of the querying procedures, but only to the extent that ensures adequate empirical validation of the developed procedures. Additionally, it would be desirable if the developed querying solutions are close to the solutions suitable for dissemination to applicants of knowledge space software tools. *Mathematica* as an integrated platform suitable for computing, programming and typesetting seemed as an appropriate candidate. Needless to say, interactive capabilities and *Mathematica*'s potential in teaching was an additional argument facilitating the selection. Unfortunately, standard packages distributed with *Mathematica* were not found sufficient for systematic explorations, research and development on the querying procedures, so a need for developing a specific package that would enable has arisen.

KSMP was conceived as a more general environment under which major concepts and findings from the knowledge space theory would be implemented, while *QEOS* application was conceived as a part of the package dealing especially with the querying experts issues. After developing the querying procedures with satisfactory properties, one possibility would be to efficiently implement the procedures using one of standard languages as *C/C++* and disseminate the built tool. Alternatively, the developed procedures could be joined with a *Mathematica* front-end interface to the applicant into an user-friendly application. *QEOS* is conceived to be such an application, aiming to facilitate building the knowledge space theory constructs in a variety of knowledge domains using querying experts methods.

Mathematica as a platform suitable for research and teaching within the field of knowledge spaces

Providing integrated powerful computing, typesetting and programming capabilities, *Mathematica* offers an attractive working environment for research within the field of knowledge space theory. Extending this with a possibility of presenting both the standard findings as well as the cutting edge results in the form of interactive notebooks to students, makes *Mathematica* recommendable. In addition to these major benefits, several other characteristics are found quite useful. Existence of mechanisms (*MathLink* and the interprocess-communication mechanism) enables integrating *Mathematica* with existing *Unix/C* knowledge space tools, allowing possibilities of combining them with functions developed in *Mathematica* or invoking them from the front-end in a transparent way. These tools are optimized for computations on structures having sizes that are found in relevant empirical applications within the field, therefore it would be desirable to have a possibility to use these tools. Additionally, possibilities of redirecting computation output to separate notebooks and automatic export to different file formats (both realizable by front-end programming) further ease various routine tasks. Probably, for the research on knowledge spaces, the most valuable part of the standard *Mathematica* distribution would be Skienna's *DiscreteMath`Combinatorica`* package [Skienna 1990 #3]. This package served as starting point for developing the *KSMP*.

Installation

Software and hardware requirements, portability and versioning

The package has been developed and tested on a Pentium 233 MHz computer equipped with 96 MB of RAM running Windows 2000 Professional operating system. *Mathematica* version 4.0 was used. No effort has been taken to check backwards compatibility with 3.x versions of *Mathematica*. In general, any hardware/operating system configuration with installed at least version 4.x *Mathematica* may be regarded as sufficient for running the package. However, an applicant has to be aware that the amount of available memory will greatly influence usability of the package for applications.

In order to ensure compatibility with the existing knowledge space tools, to facilitate exchange of data with other software products (e.g. *SPSS*) and to allow easy migration of the data between different hardware/operating systems, it was decided to use pure ascii as the format for the data files. It was decided so despite expected decrements in the package performance occurring at some points caused by such decision.

The version 1.0 is the first publicly available version of the package.

Installing the package

The directory *KnowledgeSpaces* contains all package files including documentation. This directory should be on the search path. The simplest way would be to place the directory in one of the standard locations for packages, for example in *$TopDirectory/AddOns/ExtraPackages* at system level, or alternatively at user's level in *$PreferencesDirectory/AddOns/Applications*. If from any reason it would be desirable that the package resides in a specific non-standard location (e.g. *MyPackagesDir*), this could be, for example, accomplished by issuing the command

```
PrependTo[$Path, "MyPackagesDir"]
```

The package can be autoloaded by issuing the command

```
<< KnowledgeSpaces`
```

which can also be put in *init.m* file. Autoloading the package is advised because this approach eliminates unpleasant situation that might occur when one of its functions is called before the package has been actually loaded. Alternatively, all packages modules can be loaded at any time using

```
Needs["KnowledgeSpaces`Querying`"];
Needs["KnowledgeSpaces`Assessment`"];
```

Package documentation

The package documentation consists of the technical report, user's manual and on-line documentation based on these two documents. The on-line documentation is accessible via the *Help Browser* under the *Add-ons/Knowledge Spaces* node. After installing the package, it is necessary to rebuild the help index in order to make the package on-line documentation visible. This can be achieved by choosing *Rebuild Help Index* from the *Help* menu.

Additionally, standard ways of getting assistance for built-in functions are present for all the package functions as well. Using the *?* command one can see the description of a function, for example

```
? QEOSQueryExpertOnCompetenceSpace
```

QEOSQueryExpertOnCompetenceSpace[E,SkOrExp,options] queries a human (SkOrExp represents a description of skills) or a simulated expert (SkOrExp represents a simulated expert) on structure of the set E of elementary competencies and returns the established structure either as a space or assertions. A valid option is QueryingAlgorithm (Koppen [default]). For additional valid options see QEKQueryExpertByKoppen.

The *Ctrl-Shift-K* combination may ease and speed up writing of calls using the functions.

Package design

Architecture of the package

In general, the package has been developing under assumption that it will be used by two essentially different class of users: applicants and developers. An applicant would probably be only interested in possibilities of using the package in empirical investigations, theoretical explorations, simulation studies and teaching. A developer, in contrast to an applicant, probably will also want to suit the package to his or her own needs, therefore to build own solutions, extensions and applications based on the package. Having this in mind together with a whole variety of knowledge space theory concepts that potentially might be of interest, and therefore implemented in the package, an idea of a package that would allow further developments in serveral directions independently emerged. It is believed that such design would make the package appropriate both for applicants and developers.

The package is conceived as consisting of a core, various modules and applications that form the four abstract layers of the package. The core's purpose is to ensure availability of basic data structures and functions, representing the basic knowledge space theory concepts, to the modules, as well as to ensure both flexible and uniform way of working with data files, and to provide reporting facilities. The modules are conceived to have functionalities corresponding to the different topics and approaches within the knowledge space theory. The modules are forming the layer three of the package. The layer four is reserved for applications that resides mostly on using the modules. The layers and modules are consisting of components that mostly correspond to different concepts, topics and approaches in the knowledge space theory.

In this package version, the core of the package consists of the two layers (*KSMP* layer one and *KSMP* layer two) with an additional module (*KSMP* warp core) added to the layer one. The layer three consist of the following modules: skills module, querying module and assessment module. The first application planed for the layer four is *QEOS* whose aim is to provide an user friendly interface to an applicant interested in establishing the structures by different methods of querying experts.

KSMP Layer one

The layer one consists of the following components: *Sets*, *Relations*, *Functions*, *Structures*, *DataFiles*, *Report* and *Miscellaneous* component. The aim of this layer is to be an interface between basic *Mathematica* data structures and functions at one side and functions implementing concepts and procedures from the knowledge space theory at the other.

The aim of the component *DataFiles* is to ensure reading and writing of data files both in *KSMP* native formats as well as in formats used by existing knowledge spaces tools, especially *KST* and *SRBT* tools. The aim of *Report* component is to equip both the functions from the layer two and the modules with facilities for easy producing uniform reports of interest to an applicant. The aim of the components *Sets*, *Relations* and *Functions* are to represent selected concepts from the set theory, theory of relations and theory of functions in a way suitable for basing the layer two and the modules on them, having the aims of the package as the ultimate goal. The component *Structures* represents the concept of structure, as it is defined and used within the knowledge space theory. The component *Miscellaneous* is a place for putting all public function that might be of interest mostly for a developer, which have not found their place in the other components.

The list of implemented functions is available in the glossary of *KSMP* functions.

KSMP Warp core

Warp core is an additional module to the layer one. Its purpose is to provide data structures and data files handling facilities to the package, which would enable usage of *KSMP* in relevant empirical knowledge space theory applications respecting the memory limitations of personal computers.

The list of implemented functions is available in the glossary of *KSMP* functions.

KSMP Layer two

The aims of the layer two of the *KSMP* package are to implement the selected knowledge space theory concepts, to set up an appropriate environment for conducting simulations, and finally to ensure access to the pools of items and skills for the package modules. The layer two consists of the following components: *Items and skills*, *Modeling expert*, *Modeling student*, *Theory of relations and order theory*, *Knowledge structures*, *Basis*, *Surmise systems* and *Entail relations*. The components from the layer two are supposed to be the basis for the modules, that is to serve as an interface between the core and the package modules. Such design should facilitate a separation of the core from the modules, ensuring that eventual future redesigns of the core will be reflected in the modules as less as possible.

The aim of the component *Items and skills* is to provide data structures and functions for describing items and skills that are to be used by modules while performing different procedures, for example querying an expert on structure of a set of items or assessing knowledge of a student given the structure. For additional information see the section on the used data file formats. The aim of the components *Modeling expert* and *Modeling student* is to provide data structures for describing domain metaknowledge of an expert or domain knowledge of a student. Additionally, these descriptions also provide place for uncertainty parameters representing careless errors or lucky guesses of a student or an expert, or a subjective certainty in a response or an answer given. Expert's metaknowledge can be described either using a structure or a diagnostic, but also using alternative solutions such as lists of positive, negative and undecidable assertions having different uncertainty parameters. Such modeled experts and students are to be used in simulation studies.

The aim of the components *Theory of relations and order theory*, *Knowledge structures*, *Basis*, *Surmise Systems* and *Entail relations* is to implement selected concepts from the corresponding knowledge space theory topics in the package.

The list of implemented functions is available in the glossary of *KSMP* functions.

KSMP Skills module

The aim of the Skills module is to implement various approaches for handling latent skills and competencies existing in the knowledge space theory. In this package version, only the competence-performance approach is implemented inside the *Component-performance* component of the module. The querying module, especially its component that implements querying experts on skills, depends on this module.

The list of implemented functions is available in the glossary of *KSMP* functions.

KSMP Querying module

The aim of the Querying module is to implement existing approaches for establishing various knowledge space theory constructs by querying experts. The module in this package version consists of the following components: *Koppen's algorithm*, *Querying experts on competence-performance diagnostic*, *Querying Session Sonde*, and *General querying experts* component.

The Koppen's algorithm component implements establishing of knowledge structures based on the concepts of entail relation and entailment, as well as a full version of Koppen's block-by-block algorithm as published in Koppen (1993). The Querying Experts on competence-performance diagnostic currently implements only a straightforward procedure for querying experts on diagnostic. The most important parts of this component are the procedure for querying on competence space based on Koppen's block-by-block algorithm and the straightforward procedure for querying on interpretation function. The Querying Session Sonde introduces the concept of observation sonde, whose goal is to be a method for exploring the behavior and performance of querying procedures. The General querying experts component is conceived as a component that unites all implemented querying approaches and represents an interface between the module and applications belonging to the fourth layer of the package.

The querying procedures may operate either on a human expert or on a simulated expert. Querying reports having different extent of verbosity may easily be produced due to the global verbosity variable and the verbosity option that majority of the package functions have.

The list of implemented functions is available in the glossary of *KSMP* functions.

KSMP Assessment module

The aim of the *KSMP* assessment module is to implement approaches to adaptive knowledge assessment existing within the knowledge space theory. In this package version, this module is included only for demonstration purposes and it consists only of a component for *Deterministic knowledge assessment*.

The list of implemented functions is available in the glossary of *KSMP* functions.

File formats used

KSMP supports three major data file formats: *spacefile*, *itemsfile* and *skillsfile* formats. All of them are pure ascii formats due to the intention to preserve compatibility with the existing knowledge space tools, as well as to ensure easy transfer of data between different hardware/operating systems configurations and to facilitate exchange of data with other software products.

With respect to *KSMP* design, the *Datafile* component of the package layer one is responsible for appropriate handling of data files. Since other knowledge space tools, primarily *KST* tools and *SRBT* tools, are developing in parallel and independently of *KSMP* package, some effort has been taken to equip *KSMP* with facilities for supporting possibly different versions of the mentioned file formats.

Spacefile format

The *spacefile* format is designed to store information on families of subsets, particulary knowledge structures, competence structures and examinees' observable response patterns. This format was inherited from already developed *KST* and *SRBT* tools.

The header consists of the signature of the tool that produced the file and metainformation, while the body consists of the actual family of subsets. The metainformation is given by the two lines of the header after the comment lines. It consists of two integers, representing the overall number of elements from which the subsets are created and the actual number of subsets present in the file. Further lines consist of binary string, where each line represents a subset, while binary digits stands for absence or presence of a particular element (e.g. item or elementary competence) in the subset.

Here is an example of the spacefile:

```
#
# KSMP v1.0
#
3
4
000
100
110
111
```

The spacefile above represents the following family of subsets {{}, {1}, {1, 2}, {1, 2, 3}} taken from the set {1, 2, 3}.

Itemsfile format

The *itemsfile* format is designed to store description of items for purposes of querying experts or knowledge assessment. This format is native to *KSMP*.

The header consist of the signature of a tool that produced the file. The body is given as a sequence of records, where the record describes one item. Each record consists of five lines. The first line contains code of the item. This code is used for item identification by *KSMP* functions. The next line consists of a description of the item. For example, querying module functions use this description in order to present assertions to a human expert. The next line consist of a

question text of the item that is, for example, used by knowledge assessment module functions while presenting the item to an examinee. The fourth line of each record identifies code of the correct answer for the item. The fifth line of a record consist of a list of pairs storing information on possible answers to an multiple-choice item. Each pair consists of code and text. The code identifies the answer, while the text is used by *KSMP* functions while presenting the item

This is an example of the itemsfile:

```
# KnowledgeSpaces Mathematica Package
# KSMP v1.0
1
Description 1
Question Text 1
1
{{1,"Answer 1"},{2,"Answer 2"},{3,"Answer 3"}}
2
Description 2
Question Text 2
2
{{1,"Answer 1"},{2,"Answer 2"},{3,"Answer 3"}}
```

The itemsfile above describes two items. For example, an examinee's answer "Answer 2" on the presented questions "Question Text 2" will be treated as corrected.

Skillsfile format

The *skillssfile* format is designed to store description of latent skills and competencies for purposes of querying experts module. This format is native to *KSMP*.

The header consist of the signature of a tool that produced the file. The body is given as a sequence of records, where each record describes one latent skill or elementary competency. The record consists of three lines. The first line contains code of a skill. This code is used for skill identification by *KSMP* functions. The second line consist of the skill name, while the third line consists of a description of the skill. For example, querying module functions use the skill names while presenting assertions to a human expert.

This is an example of the skillsfile:

```
# Skills File
# KSMP v1.0
1
Skill_1
Description_Skill_1
2
Skill_2
Description_Skill_2
3
Skill_3
Description_Skill_3
```

Further developments

Further developments of the package may be classified into three overlapping categories: development of the core, development of the modules, and development of applications.

In this package version, a plenty of written functions perform well, given the memory limitations of personal computers, only on small sets of items, that is on the sets whose cardinalities do not reach numbers requested by relevant empirical investigations. Therefore, efforts will be taken in order to equip the package with data representations, auxilary functions and swapping file mechanisms that will ensure adequate performance of the functions on the sets having the relevant sizes for empirical investigations. Probably, the future versions of the package will include two operating modes of the package: *research mode* and *application mode*. Research mode will be suitable for research, teaching, explorations and simulations on small sets of items, while the application mode will be suitable for usage in empirical investigations. The *KSMP* Warp Core addition to the layer one represents an effort in this direction. Besides the further development of the warp core itself, the other functions belonging to the package core will be also upgraded to support the warp core solutions. In addition, a mechanism for automatic selection between research and application mode, based primarily on the available memory, will also be implemented.

Another category of developments consists of extending and enhancing the existing modules, as well as of developing additional modules. Extending the existing modules will include implementing other approaches to skills as additional components inside the *KSMP* Skills module, and implementing other querying procedures as additional components inside the *KSMP* Querying module.

The last, but not the least, category of further developments consists of developing applications that can be seen as members of the abstract layer four of the package. Within the *QEOS* project, there is a plan to equip the layer four with the *QUEOS* application, whose aim is to provide an user friendly environment for establishing the knowledge space theory constructs using different querying procedures. Namely, after developing the querying procedures with satisfactory properties, one possibility would be to efficiently implement the procedures using one of standard languages as *C/C++* and disseminate the built tool. However, the alternative is to develop *QEOS* application having a *Mathematica* front-end interface to applicants. Such application would enable easy usage of the developed querying procedures, and in turn, building of the structures in a variety of knowledge domains.

Bibliography

Albert, D. and J. Lukas (1999). Knowledge Spaces: Theories, Empirical Research, and Applications. Mahwah, New Jersey and London, Lawrence Erlbaum Associates, Publishers.

Doignon, J.-P. and J.-C. Falmagne (1999). Knowledge Spaces. Berlin, Springer-Verlag.

Koppen, M. (1993). "Extracting Human Expertise for Constructing Knowledge Spaces: An Algorithm." Journal of Mathematical Psychology 37: 1-20.

Korossy, K. (1999). Modeling Knowledge as Competence and Performace. In D. Albert and J. Lukas, Knowledge Spaces: Theories, Empirical Research and Applications. Mahwah, NJ and London, Lawrence Erlbaum Associates, Publishers: 103-132.

Maeder, R. E. (1997). Programming in Mathematica. Reading, MA, Addison-Wesley Publishing Company.

Skienna, S. (1990). Implementing discrete Mathematics: combinatorics and graph theory with Mathematica. Redwood City, CA, Addison-Wesley Publishing Co.

Skienna, S. (1998). The algorithm design manual. New York, Springer-Verlag.

Wolfram, S. (1999). The Mathematica Book, Cambridge University Press.

Glossary of *KSMP* functions

version 1.0

This glossary contains all public functions of the *KSMP* package listed alphabetically. The majority of functions are of interest both to applicants and developers, while some functions are probably of interest only to a developer.

Atoms

Atoms[SF] returns all subfamilies from the family forming the structure SF that are atoms by at least one element from the set forming the structure.

AtomsAt

AtomsAt[SF,x] returns all atoms at the element xœS given the structure SF .

BasisFromSpace

BasisFromSpace[SF] returns the basis of the space SF.

BasisToSpace

BasisToSpace[F] returns the space constructed from the given basis F.

CommentLines

CommentLines[SFile] returns the list of all line numbers from the datafile object SFile that contain comments.

CompetenceSpaceQ

CompetenceSpaceQ[EK] returns True iff the structure EK is a competence space.

CompetenceStructureQ

CompetenceStructureQ[EK] returns True iff the structure EK is a competence structure.

CompositionOfFunctions

CompositionOfFunctions[{fn,..,f1}] returns the composition (fn o .. o f1) of the functions fn,..,f1 under assumption that such composition is defined.

CPADiagnosticCompetenceSpace

CPADiagnosticCompetenceSpace[Diag] returns the competence space forming the given diagnostic Diag.

CPADiagnosticElementaryCompetencies

CPADiagnosticElementaryCompetencies[Diag] returns the set of elementary competencies forming the given diagnostic Diag.

CPADiagnosticFromInterpretationFunction

CPADiagnosticFromInterpretationFunction[k] returns the (6-tupel) diagnostic induced by the interpretation function k. It is assumed that the competence structure given by k is based on a set of elementary competencies.

CPADiagnosticInterpretationFunction

CPADiagnosticInterpretationFunction[Diag] returns the interpretation function forming the given diagnostic Diag.

CPADiagnosticItems

CPADiagnosticItems[Diag] returns the set of items forming the given diagnostic Diag.

CPADiagnosticPerformanceSpace

CPADiagnosticPerformanceSpace[Diag] returns the performance space forming the given diagnostic Diag.

CPADiagnosticRepresentationFunction

CPADiagnosticRepresentationFunction[Diag] returns the representation function forming the given diagnostic Diag.

CPAEqualDiagnosticsQ

CPAEqualDiagnosticsQ[Diag1,Diag2] returns True iff the diagnostics Diag1 and Diag2 are equal.

CPAInterpretationFunctionAxiomsQ

CPAInterpretationFunctionAxiomsQ[k] returns True iff the function k satisfies the axiom 1 of interpretation functions.

CPAInterpretationFunctionConformTo

CPAInterpretationFunctionConformTo[k] returns the interpretation function obtained from the given interpretation function k after reducing its set of items to a competence-based item set.

CPAInterpretationFunctionInitialize

CPAInterpretationFunctionInitialize[K] returns the empty interpretation function given the competence space K and the empty set of problems.

CPAInterpretationFunctionInitialize[A,K] returns the initialized interpretation function consisting of the non-interpreted items from the set A of items given the competence space K.

CPAInterpretationFunctionInterpretations

CPAInterpretationFunctionInterpretations[k] returns the list of interpretations contained in the interpretation function k.

CPAInterpretationFunctionItems

CPAInterpretationFunctionItems[k] returns the set of items contained in the interpretation function k.

CPAInterpretationFunctionQ

CPAInterpretationFunctionQ[f] returns True iff the function f is an interpretation function.

CPAInterpretationFunctionUpgrade

CPAInterpretationFunctionUpgrade[k,I] returns the interpretation function k upgraded with the (possibly empty) list I of interpretations or an individual interpretation I.

CPAInterpretationOfItem

CPAInterpretationOfItem[k,x] returns the interpretation of the item x given the interpretation function k.

CPAInterpretationsDeleteItemInterpretations

CPAInterpretationsDeleteItemInterpretations[KXs,q] returns the list of interpretations after deleting all interpretations on the item q from the list KXs of interpretations.

CPAInterpretationsInterpretationAdd

CPAInterpretationsInterpretationAdd[KXs,kx] returns the interpretations KXs after adding the interpretation kx.

CPAInterpretationsInterpretationReplace

CPAInterpretationsInterpretationReplace[KXs,NewKx] returns the interpretations KXs after replacing the corresponding interpretation with the interpretation NexKx.

CPAMakeDiagnostic

CPAMakeDiagnostic[K,A,P,k,p] returns the data structure representing the diagnostic given by the family forming the competence structure K, the set A of items, the family forming the performance structure P, the interpretation function k and the representation function p.

CPAMakeDiagnostic[E,K,A,P,k,p] returns the data structure representing the diagnostic given by the set E of elementary competencies, the family forming the competence structure K,the set A of items, the family forming performance structure P, the interpretation function k and the representation function p.

CPAMakeInterpretationFunction

CPAMakeInterpretationFunction[A,K,KXs] returns the interpretation function given the set A of items, the competence structure K, and the mapping KXs interpreting each item by assigning competence states.

CPAMakeRepresentation

CPAMakeRepresentation[C,Items] returns the data structure containing the representation formed by the given competence state C and the set Items.

CPAMakeRepresentationFunction

CPAMakeRepresentationFunction[K,A,PCs] returns the data structure containing the representation function formed from the competence structure K, the set A of items and the collection PCs of representations.

CPARepresentationCompetenceState

CPARepresentationCompetenceState[PC] returns the competence state of the given representation PC.

CPARepresentationFunctionCompetenceStructure

CPARepresentationFunctionCompetenceStructure[p] returns the competence structure forming the given representation function p.

CPARepresentationFunctionFromInterpretationFunction

CPARepresentationFunctionFromInterpretationFunction[k] returns the representation function given the interpretation function k.

CPARepresentationFunctionRepresentations

CPARepresentationFunctionRepresentations[p] returns the collection of representations forming the representation function p.

CPARepresentationItems

CPARepresentationItems[pC] returns the set of items for the given representation pC.

CPARepresentationOfCompetenceState

CPARepresentationOfCompetenceState[k,C] returns the data structure {C,p(C)}, where p(C) is a set of items assigned by the interpretation function k to the given competence state C.

CPARepresentations

CPARepresentations[k] returns the list of all representations of the competence structure derived from the interpretation function k.

CPARepresentationStructureFromInterpretationFunction

CPARepresentationStructureFromInterpretationFunction[k] returns the induced performance structure from the given interpretation function k.

CPARepresentationStructureQ

CPARepresentationStructureQ[AP,k] returns True iff the given performance structure AP is the representation structure under the interpretation function k.

CSAxiom1Q

CSAxiom1Q[EK] returns True iff the structure EK fulfills the property $\bigcup K=E$.

CSAxiom2Q

CSAxiom2Q[EK] returns True iff the structure fulfills the property that the family K contains both the empty set and the set E.

CSAxiom3Q

CSAxiom3Q[EK] returns True iff the structure EK is stable under union.

DataFileBodyExtract

DataFileBodyExtract[SFL,FileFormat] returns the extracted body from the non-commented data list SFL representing the datafile object using the file format FileFormat.

For the supported file formats see FileFormatsInfo.

DataFileBodyInterpret

DataFileBodyInterpret[B,FF] interprets the extracted body B from the datafile object in the FF format.

DataFileCommentDrop

DataFileCommentDrop[SFile] returns the list representing the datafile object SFile after removing comments.

DataFileCommentExtract

DataFileCommentExtract[SFile] returns the list containing comments from the datafile object SFile.

DataFileCommentPrint

DataFileCommentPrint[DF] extracts and prints the comments from the datafile object DF.

DataFileHeadExtract

DataFileHeadExtract[SFL,FileFormat] returns the extracted head the from the non-commented data list SFL representing the datafile object in accordance with the datafile format FileFormat.

For the supported datafile formats see FileFormatsInfo.

DataFileHeadInterpret

DataFileHeadInterpret[H,FF] interprets the extracted head H from a data file object in the FF format.

DataFileLength

DataFileLength[SFile] returns the number of lines in the datafile object SFile.

DataFileRead

DataFileRead[Filename] creates a datafile object representing the data file Filename.

DataFileWrite

DataFileWrite[FN,M,"KST_SpaceFile"] writes the matrix M, together with the derived head, in the datafile FN using the KST_SpaceFile format.

DataFileWrite[FN,Items,"KSMP_ItemsFile"] writes the items description Items into the file FN using the KSMP_ItemsFile format.

DataFileWrite[FN,Skills,"KSMP_SkillsFile"] writes the skills description Skills into the file FN using the format KSMP_SkillsFile.

ElementsInRelationQ[SR,a,b] returns True iff the elements a and b are in the relation SR.

EntailmentFromEntailRelation

EntailmentFromEntailRelation[AAPB] returns the entailment derived from the entail relation AAPB.

EntailmentToEntailRelation

EntailmentToEntailRelation[AAPx] returns the entail relation derived from the entailment AAPx.

EntailRelationFromKnowledgeSpace

EntailRelationFromKnowledgeSpace[SF] returns the entail relation given the knowledge space SF.

EntailRelationToKnowledgeSpace

EntailRelationToKnowledgeSpace[ESR] returns the knowledge space given the entail relation ESR.

EqualFunctionsQ

EqualFunctionsQ[f1,f2] returns True iff the functions f1 and f2 are equal.

EqualRelationsQ

EqualRelationsQ[SR1,SR2] returns True iff the relations SR1 and SR2 are equal.

EqualSetsQ

EqualSetsQ[A,B] returns True iff the sets A and B are equal.

EqualStructuresQ

EqualStructuresQ[SF1,SF2] returns True iff the structure SF1 is equal to the structure SF2.

ExistOneQ

ExistOneQ[X,pf] returns True iff the property pf holds for at least one element of the set X.

ExpertAnswerCertaintyFactor

ExpertAnswerCertaintyFactor[Ans] returns the certainty factor of an expert's answer Ans.

ExpertAnswerLogicalValue

ExpertAnswerLogicalValue[Ans] returns the logical value of the expert's answer Ans.

ExpertCarelessErrors

ExpertCarelessErrors[E] returns the uncertainty that the expert E makes a careless error during a querying session.

ExpertMakeAnswer

ExpertMakeAnswer[LV,CF] returns the data structure describing an expert's answer consisting of the logical value LC and the certainty factor CF.

ExpertMetaKnowledge

ExpertMetaKnowledge[E] returns a domain metaknowledge of the expert E.

FamilyQ

FamilyQ[F] returns True iff SF is a family of subsets.

FileFormatsInfo

FileFormatsInfo[FileFormat,Info] returns the information Info about the requested file format FileFormat. Currently, supported formats are: "KST_SpaceFile" and "KST_PatternFile". The stored information on the formats is: "Head".

ForAllPairsQ

ForAllPairsQ[X,pf] returns True iff the predicate pf gives True for all ordered pairs from X.

ForAllPairsQ[X,Y,pf] returns True iff the predicate pf gives True for all pairs from X x Y.

ForAllQ

ForAllQ[X,pf] returns True iff the property pf holds for all elements of the set X.

ForAllTriplesQ

ForAllTriplesQ[X,pf] returns True iff the predicate pf gives True for all ordered triples from X.

ForAllTriplesQ[X,Y,Z,pf] returns True iff the predicate pf returns True for all the ordered triples from X x Y x Z.

FunctionBijectionQ

FunctionBijectionQ[f] returns True iff the function f is a bijection.

FunctionChangeCodomain

FunctionChangeCodomain[f,C1] is a primitive function that returns the function f after replacing its codomain with the new set C1.

FunctionChangeDomain

FunctionChangeDomain[f,D1] is a primitive function that returns the function f after replacing its domain with the new set D1.

FunctionChangeMapping

FunctionChangeMapping[f,M1] is a primitive function that returns the function f after replacing its mapping rule with the new mapping rule M1.

FunctionCodomain

FunctionCodomain[f] returns the codomain of the function f.

FunctionDomain

FunctionDomain[f] returns the domain of the function f.

FunctionInjectionQ

FunctionInjectionQ[f] returns True iff the function f is an injection.

FunctionMapping

FunctionMapping[f] returns the mapping rule of the function f.

FunctionMappingImage

FunctionMappingImage[f] returns the function's image calculated using the mapping rule of the function f.

FunctionMappingOriginal

FunctionMappingOriginal[f] returns the function's original calculated using the mapping rule of the function f.

FunctionQ

FunctionQ[F] returns True iff F represents a function.

FunctionRestriction

FunctionRestriction[f,D1] returns the function f restricted to its subdomain D1.

FunctionSubsetImage

FunctionSubsetImage[f,D1] returns the image of the subset D1 under the function f.

FunctionSubsetImage[functions,D1] returns the image of the subset D1 after applying a list of functions (from right to left).

FunctionSurjectionQ

FunctionSurjectionQ[f] returns True iff the function f is a surjection.

FunctionValue

FunctionValue[f,x] returns the function value f(x) of the element x.

FunctionValue[functions,x] returns the value of x after applying the list of functions (from right to left).

FunctionValue[f,D1] returns values of the elements from the set D1 after applying the function f.

FunctionValue[functions,D1] returns values of the elements from the set D1 after applying the list of functions (from right to left).

FunctionWellDefinedQ

FunctionWellDefinedQ[f] returns True iff the original of a mapping rule of the function f is equal to the domain of the function.

FX

FX[SF,s] returns all members of the family forming the structure SF, which contain the element s of the set forming the structure.

FX[SF,C,NC] returns all members of the family forming the structure SF, which contain all the elements from the set C and have empty intersection with the set NC.

FX[SF,C] returns all members of the family forming the structure SF, which contain all the elements from the set C.

FXFilterOrderDown

FXFilterOrderDown[SF,f] returns all members of the family forming the structure SF, which are subsets of the family member f.

FXFilterOrderUp

FXFilterOrderUp[SF,f] returns all members of the family forming the structure SF, which are supersets of the family member f.

GenerateFamilyByTakingUnions

GenerateFamilyByTakingUnions[F] returns the family generated by taking all possible unions of the elements from the family F.

IAssertionCompetenceState

IAssertionCompetenceState[IAss] returns the competence state of the interpretation assertion IAss.

IAssertionItem

IAssertionItem[IAss] returns the item of the interpretation assertion IAss.

ItemAnswers

ItemAnswers[Item] returns the list of all admissible answers on the item Item, where each answer consists of a code and a description.

ItemCode

ItemCode[Item] returns the code of the given Item.

ItemCorrectAnswer

ItemCorrectAnswer[Item] returns the correct answer of the given Item.

ItemDescription

ItemDescription[Item] returns the description of the given Item.

ItemGet

ItemGet[Items,Code] returns the first item in the description Items whose code matches the code Code.

ItemText

ItemText[Item] returns the textual question of the given Item.

KAAskStudent

KAAskStudent[Stud,QE,"Simulation"] returns an answer from the simulated student Stud on the question QE.

KAAskStudent[Items,QE,"Human"] returns an answer from a human student on the question QE described in Items.

KADAssessKnowledge

KADAssessKnowledge[SF,ItemsStudent,options] performs the deterministic assessment of a student's knowledge given the knowledge structure SF. In case of a human student (option Examinee, value "Human" [default]) the parameter ItemsStudent takes the description of items, while in case of a simulated student (option Examinee, value "Simulation") it stores a simulated student. The function may return either a knowledge state (option AssessmentResult, value "State" [default]) or an assessment history (option AssessmentResult, value "History"). The additional valid option is Verbosity (Off [default],1,5,10,20).

KADSelectNextQuestion

KADSelectNextQuestion[SF,{CA,NCA}] returns the next item to be asked by the deterministic knowledge assessment procedure on the knowledge structure SF assuming already collected correct CA and incorrect NCA answers by a student.

KADSelectNextQuestion[SF] assumes that no previous answers have been collected.

KnowledgeSpaceQ

KnowledgeSpaceQ[SF] returns True iff the structure SF satisfies formal requirements of a knowledge space.

KnowledgeSpaceQuasiOrdinalQ

KnowledgeSpaceQuasiOrdinalQ[SF] returns True iff the structure SF satisfies formal requirements of a knowledge space.

KnowledgeStructureQ

KnowledgeStructureQ[SF] returns True iff the structure SF satisfies formal requirements of a knowledge structure.

KSAxiom1

KSAxiom1[SF] returns True iff the structure SF contains both the empty set and the whole set S.

KSAxiom2C

KSAxiom2C[SF] returns True iff the structure SF is closed under union.

KSAxiom2S

KSAxiom2S[SF] returns True iff the structure SF is stable under union.

KSAxiom3C

KSAxiom3C[SF] returns True iff the structure SF is closed under intersection.

KSAxiom3S

KSAxiom3S[SF] returns True iff the structure SF is stable under intersection.

KSMPBinaryToList

KSMPBinaryToList[BL] returns the list of positive integers calculated from the binary list BL.

KSMPBinaryToWarp

KSMPBinaryToWarp[BL] returns the warp representation of the binary representation list BL representing a list of positive integers.

KSMPListToBinary

KSMPListToBinary[L] returns the binary representation list of the list L consisting of positive integers.

KSMPListToWarp

KSMPListToWarp[L] returns the warp representation of the list L of positive integers.

KSMPWarpToBinary

KSMPWarpToBinary[W] returns the binary list representation of the given warp W.

KSMPWarpToList

KSMPWarpToList[W] returns the list of positive integers from its warp representation W.

LineCommentQ

LineCommentQ[L] returns True iff the line L of a datafile object contains a comment.

MakeExpert

MakeExpert[MK,CE] returns the data structure representing an expert consisting of the domain's metaknowledge MK and the uncertainty CE of making a careless error during a querying session.

MakeFunction

MakeFunction[D,C,M] creates a function from the set D to the set C using the mapping rule M.

MakeIAssertion

MakeIAssertion[CS,Item] returns the interpretation assertion consisting of the competence state CS and the item Item.

MakeItem

MakeItem[Code,Desc,QText,CorrAns,LofDist] returns a data structure representing the item consisting of the unique identifier Code, description Desc, text QText of a question, correct Answer CorrAns, and the list LofDist of admissible answers, where each answer consists of a code and a textual description.

MakeKnowledge

MakeKnowledge[QK,"Items"] returns a data structure representing a student's domain knowledge given as the set QK of items.

MakeMetaKnowledge

MakeMetaKnowledge[Diag,Diagnostic] returns a data structure representing an expert's metaknowledge, in terms of the competence-performance approach, using the diagnostic Diag.

MakeRelation

MakeRelation[S,R] returns a data structure representing the relation given by the set S and the subset R of the cartesian product S×S.

MakeRelation[S1,S2,R] returns a data structure representing the relation from the set S1 to the set S2 given by the subset R of the cartesian product S1×S2.

MakeSimpleExpert

MakeSimpleExpert[Diag] returns an ideal expert whose domain metaknowledge is given by the competence-performance diagnostic Diag.

MakeSimpleStudent

MakeSimpleStudent[QK] returns an ideal student whose domain knowledge is given by the set QK of items.

MakeSkill

MakeSkill[Code,Name,Desc] returns the data structure representing a skill consisting of the unique identifier Code, the name Name and the description Desc.

MakeStructure

MakeStructure[X,F] returns a data structure representing the structure consisting from the set X and the family F of subsets.

MakeStudent

MakeStudent[QK,CE,LG] returns the data structure representing a student whose domain knowledge is given by the set QK of items the student is capable of solving, the uncertainty CE of making a careless error, and the uncertainty LG of making a lucky guess during a knowledge assessment.

MakeSurmiseFunction

MakeSurmiseFunction[Q,LofCl] returns the data structure representing the surmise function given by the set Q of items and the mapping LofCl of clauses.

MakeSurmiseSystem

MakeSurmiseSystem[Q,LofCl] returns the data structure representing a surmise system given by the set Q of items and the mapping LofCl of clauses, under assumption that these objects fulfill the necessary formal requirements.

Matrix2DataFileBody

Matrix2DataFileBody[M,"KST_SpaceFile"] returns the datafile object body from the matrix M using KST_SpaceFile format.

Matrix2DataFileHead

Matrix2DataFileHead[M,"KST_SpaceFile"] returns the head of a datafile object from the matrix M using KST_SpaceFile format.

MaximalsOfRelation

MaximalsOfRelation[SR] returns all maximals of the relation SR.

MetaKnowledgeAssertionsNegative

MetaKnowledgeAssertionsNegative[Assertions] returns all negative assertions contained in the expert's metaknowledge Assertions.

MetaKnowledgeAssertionsPositive

MetaKnowledgeAssertionsPositive[Assertions] returns all positive assertions contained in the expert's metaknowledge Assertions.

MetaKnowledgeAssertionsRest

MetaKnowledgeAssertionsRest[Assertions] returns all neither positive nor negative assertions contained in the expert's metaknowledge Assertions.

MinimalsOfRelation

MinimalsOfRelation[SR] returns all minimals of the relation SR.

NonCommentLines

NonCommentLines[SFile] returns the list of all line numbers of the datafile object that do not contain a comment.

PerformanceSpaceQ

PerformanceSpaceQ[AP] returns True iff the structure AP is a performance space.

PerformanceStructureQ

PerformanceStructureQ[AP] returns True iff the structure AP is a performance structure.

PowerSet

PowerSet[S] returns the powerset of the set S. By default ranking, all subsets with the cardinality k-1 appear before the subsets with the cardinality k.

PQxPQ

PQxPQ[Q] returns the cartesian product on the powerset of the set Q from which the empty set was substracted.

PQxQ

PQxQ[Q] returns the cartesian product between the set Q and its powerset from which the empty set was substracted.

PSAxiom1Q

PSAxiom1Q[AP] returns True iff the structure AP fulfills the property $\bigcup P=A$.

PSAxiom2Q

PSAxiom2Q[AP] returns True iff the structure fulfills the property that the family P contains both the empty set and the set A as its subsets.

PSAxiom3Q

PSAxiom3Q[AP] returns True iff the structure AP is stable under union.

QEKAllAssertionsToSpace

QEKAllAssertionsToSpace[Q,PosAss] returns the knowledge space derived from the positive assertions PosAss.

QEKAskExpert

QEKAskExpert[Q,Items,Assertion,Human] returns an answer from a human expert on the assertion Assertion containing some items from the set Q described in Items.

QEKAskExpert[Assert,Exp,Simulation] returns an answer from the simulated expert Exp on the assertion Assert. The expert's domain metaknowledge may be represented either using a structure or a diagnostic.

QEKAssertionConsequence

QEKAssertionConsequence[Assertion] returns the consequence contained in the assertion Assertion.

QEKAssertionPremise

QEKAssertionPremise[Assertion] returns the premise contained in the assertion Assertion.

QEKAssertionsAxiom1Drop

QEKAssertionsAxiom1Drop[Ass] returns the list of assertions excluding the trivial assertions, according to the axiom 1 of the entailment concept.

QEKCollectInferencesForNegativeAnswer

QEKCollectInferencesForNegativeAnswer[Q,PosAss,Ass]returns the data structure {P,N} consisting of positive inferences P and negative inferences N when a negative answer on the assertion Ass is collected. The set Q of items and the list PosAss of established positive assertions are given in advance. The inference rule is implemented according to Koppen (14d).

QEKCollectInferencesForPositiveAnswer

QEKCollectInferencesForPositiveAnswer[Q,PosAss,NegAss,Ass]returns the data structure {P,N} consisting of positive inferences P and negative inferences N when a positive answer on the assertion Ass is collected. The set Q of items, lists PosAss and NegAss of established positive and negative assertions are given in advance. The inference rule is implemented according to Koppen (14a),(14b),(14c).

QEKCountInferences

QEKCountInferences[Q,PosAss,NegAss,Ass] returns the data structure {Neg,Pos} containing the number Neg of inferences that would be drawn in case a negative answer is collected on the assertion Ass, and the number Pos of drawn inferences in case of a positive answer. The set Q of items, the positive inferences PosAss and negative inferences NegAss are given in advance.

QEKEntailmentFromQueryHistory

QEKEntailmentFromQueryHistory[Q,QueryHistory] returns all positive judgments from the querying history QueryHistory capturing an establishment of a structure on the set Q of items.

QEKEquivalentSubsetsQ

QEKEquivalentSubsetsQ[PosAss,A,B] returns True iff the subsets A and B are equivalent with respect to the positive inferences PosAss.

QEKEstablishKnowledgeSpaceStraightforward

QEKEstablishKnowledgeSpaceStraightforward[Q,Items,Human] returns a structure on the items Q described in Items, established by querying a human expert.

QEKEstablishKnowledgeSpaceStraightforward[Q,Expert,Simulation] returns a structure on the items Q obtained by querying the simulated expert Expert.

QEKExtractNegativeJudgments

QEKExtractNegativeJudgments[QH] returns all negative judgments from the querying history QH.

QEKExtractPositiveJudgments

QEKExtractPositiveJudgments[QH] returns all positive judgments from the querying history QH.

QEKFormulateQuery

QEKFormulateQuery[Q,Items,Assertion] returns the query question on the given assertion Assertion formulated for a human expert, given the set Q of items described in Items.

QEKInferences14a

QEKInferences14a[PosAss,Ass] returns the positive inferences contributed by the inference rule Apx&BPA&AxPy->Bpy when the assertion Ass is positively judged (APx) and added to the previous positive assertions PosAss.

QEKInferences14b

QEKInferences14b[PosAss,NegAss,Ass] returns the negative inferences contributed by the inference rule APx&AxPB&ANy->BNy when the assertion APx is added to the previous positive PosAss and negative NegAss assertions.

QEKInferences14c

QEKInferences14c[Q,PosAss,NegAss,Ass] returns the negative inferences contributed by the inference rule APx&ByPA&BNx->BNy when the assertion APx is added to the previous positive PosAss and negative NegAss assertions, given the set Q of items.

QEKInferences14d

QEKInferences14d[Q,PosAss,Ass] returns the negative inferences contributed by the inference rule ANx&APB&ByPx->BNy if ANx is added to the previous negative assertions NegAss, given the set Q of items.

QEKInitializeFirstBlock

QEKInitializeFirstBlock[Q] returns the data structure {OQ,P,N}, representing the first block in Koppen's algorithm, and consisting of open questions OQ, positive assertions P and negative assertions N.

QEKMakeAssertions

QEKMakeAssertions[Q,Premise] returns all possible assertions given the premise Premise and the set Q of items, assuming that only one element in the consequence of the assertion is present.

QEKMakeFullBlock

QEKMakeFullBlock[Q,k] returns the block (list of the assertions) of the subsets-by-items table defined by the given premise size k and the set Q of items.

QEKMakeSbITable

QEKMakeSbITable[Q] returns the subsets-by-items table given the set Q of items.

QEKOutput

QEKOutput[Q,PosAss,NegAss,options] returns the output of a querying process either as a knowledge space or as assertions depending on the OutputStructure option. The querying process is captured by the positive assertions PosAss and the negative assertions NegAss, given the set Q of items. The OutputStructure option accepts the following values: SpaceStructure (default) and Assertions.

QEKQueryExpertByKoppen

QEKQueryExpertByKoppen[Q,ItOrSkOrExp,options] queries an expert using Koppen's (1993) block-by-block algorithm. A human (ItOrSkOrExp represents a description of either items or skills) or a simulated expert (ItOrSkOrExp represents a simulated expert) is queried on the set Q of items, and a structure (either space or assertions) is returned. The valid options are: Expert (Human [default] or Simulation), OutputStructure (SpaceStructure [default] or Assertions), Verbosity (Off [default] or 1,2), QueryOn (KnowledgeSpace [default] or CompetenceSpace), and QSRReport (Result [default], Report or Both).

QEKQueryExpertStraightforward

QEKQueryExpertStraightforward[Q,Items,Human] returns the judgments obtained while querying a human expert on structure of the items Q described in Items.

QEKQueryExpertStraightforward[Q,Expert,Simulation] returns the judgments obtained while querying the simulated expert Expert on structure of the items Q.

QEKQueryTexts

QEKQueryTexts[PremiseSize,TextPart] returns the partial text of a query on an assertion given the text part TextPart (introduction or question) and the premise size PremiseSize (1,2,generic,previous_to_last).

QEKSelectionRuleMaximin

QEKSelectionRuleMaximin[Q,PosAss,NegAss,OpenQ] returns the assertion which implies a maximal number of inferences according to the maximin selection rule.

QEKSelectionRuleWeightedSums

QEKSelectionRuleWeightedSums[Q,PosAss,NegAss,OpenQ,options] returns the assertion which implies a maximal number of inferences according to the weighted sums selection rule. The option ProbabilityYES is supported having 0.5 as the default value.

QEKSelectNextQuestion

QEKSelectNextQuestion[Q,Pos,Neg,OpenQ,options] returns the data structure {NextAss,NegAnsInf,PosAnsInf} where NextAss is the next assertion for querying accompanied with its (both positive and negative) inferences NegAnsInf (in case of a negative answer) and PosAnsInf (in case of a positive answer). The selection is conducted on the set Q of items, assuming previously collected positive inferences Pos, negative inferences Neg, and the items OpenQ left open. The selection rule is given by the option SelectionRule that accepts the following values: Maximin (default) and WeightedSums.

QEKStar

QEKStar[PosAss,A] returns the A-star set of the premise A given the positive inferences PosAss.

QEOSAskExpertOnInterpretation

QEOSAskExpertOnInterpretation[K,Skills,Items,IAss,Human] returns an answer on the interpretation assertion IAss from a human expert given the competence structure K described in Skills and the interpretation assertion item described in Items.

QEOSAskExpertOnInterpretation[IAss,Expert,Simulation] returns an answer on the interpretation assertion IAss from the simulated expert Expert.

QEOSFormulateQueryInterpretations[a,Items,C,K,Skills] returns the query question on the given item a described in Items given the competence state C of the competence structure K described in Skills.

QEOSFormulateQuerySkills

QEOSFormulateQuerySkills[E,Skills,Assertion] returns the query question based on the given assertion Assertion given the set E of (elementary) skills described in Skills.

QEOSInterpretationFunctionCollectInferencesForNegativeAnswer

QEOSInterpretationFunctionCollectInferencesForNegativeAnswer[OpenQ,IAss] returns all interpretation assertion inferences given the list OpenQ of open interpretation assertions assuming the interpretation assertion IAss was negatively judged.

QEOSInterpretationFunctionCollectInferencesForPositiveAnswer

QEOSInterpretationFunctionCollectInferencesForPositiveAnswer[OpenQ,IAss] returns all interpretation assertion inferences given the list OpenQ of open interpretation assertions assuming the interpretation assertion IAss was positively judged.

QEOSInterpretationFunctionCountInferences

QEOSInterpretationFunctionCountInferences[IASSes,IAss] returns the data structure {Neg,Pos} containing the number Neg of inferences drawn if a negative answer on the interpretation assertion IAss has been collected, and the number Pos of inferences drawn in case a positive answer has been collected. The list IASSes of open interpretation assertions is given in advance.

QEOSInterpretationFunctionSelectionRuleMaximin

QEOSInterpretationFunctionSelectionRuleMaximin[OpenQ] returns the interpretation assertion which implies a maximal number of inferences according to the maximin selection rule. The list OpenQ of open interpretation assertions is given.

QEOSInterpretationFunctionSelectionRuleWeightedSums

QEOSInterpretationFunctionSelectionRuleWeightedSums[OpenQ] returns the interpretation assertion which implies a maximal number of inferences according to the weighted sums selection rule. The list OpenQ of open interpretation assertions is given.

QEOSInterpretationsFromIAssertions

QEOSInterpretationsFromIAssertions[IASSes] returns the list of interpretations derived from the list IASSes of interpretation assertions.

QEOSInterpretationsOnItem

QEOSInterpretationsOnItem[IASSes,q] returns all interpretation assertions given the item q and the list IASSes of interpretation assertions.

QEOSQueryExpertOnCompetenceSpace

QEOSQueryExpertOnCompetenceSpace[E,SkOrExp,options] queries a human (SkOrExp represents a description of skills) or a simulated expert (SkOrExp represents a simulated expert) on structure of the set E of elementary competencies and returns the established structure either as a space or assertions. A valid option is QueryingAlgorithm (Koppen [default]). For additional valid options see QEKQueryExpertByKoppen.

QEOSQueryExpertOnDiagnostic

QEOSQueryExpertOnDiagnostic[E,A,DescOrExp,opts] queries a human expert (DescOrExp having the form {Skills,Items} describes skills and items) or a simulated expert (DescOrExp represents a simulated expert whose metaknowledge is given as a diagnostic) on competence-performance diagnostic. The set A of items and the set E of elementary competencies are given. The valid options are: Expert (Human [default] or Simulation), QSRReport (Result [default], Report or Both). For other valid options see: QEOSQueryExpertOnCompetenceSpace and QEOSQueryExpertOnInterpretationFunction.

QEOSQueryExpertOnInterpretationFunction

QEOSQueryExpertOnInterpretationFunction[K,A,DescOrExp,opts] establishes an interpretation function by querying an expert given the set A of items and the competence space K. After the querying only the competence-based items set, together with its interpretations, is kept. In case of a human expert, DescOrExp has the form {Skills,Items}, where the used skills are described in Skills and the items in Items. In case of a simulated expert, DescOrExp describes the expert. The valid options are: Verbosity (Off [Default], 1,5,10,15), QSRReport (Result [default], Report or Both), Expert (Human [default], Simulation), SelectionRule (Maximin [default], WeightedSums) and RuleScope (Block [default], All).

QEOSQueryTextsInterpretations

QEOSQueryTextsInterpretations[NofSkils,TextPart] returns the textual part of a query on interpretation given the text part TextPart (introduction, middle, ending) and the number NofSkills (1,2, generic, all_minus_one) of skills.

QEOSQueryTextsSkills

QEOSQueryTextsSkills[PremiseSize,TextPart] returns the textual part of a query on dependencies among skills, given the text part TextPart (introduction or question) and the premise size PremiseSize (1,2,generic,all_minus_one).

QEOSSelectNextInterpretationQuestion

QEOSSelectNextInterpretationQuestion[OpenQ,options] returns the next interpretation assertion for querying given the (sorted) list OpenQ of open interpretation assertions respecting the given options. The valid options are: OpenQuestions (Sorted [default],Unsorted) - in case the list OpenQ is not sorted, it will be sorted with respect to the cardinality of competence states; RuleScope (Block [default], All) - whether to search for the next interpretation assertion only in the next block (defined by the cardinality of competence states) or to search all the interpretation assertions in OpenQ; SelectionRule (Maximin [default], WeightedSums) - which selection rule is to be used.

QEQueryExpertOnKnowledgeSpace

QEQueryExpertOnKnowledgeSpace[Q,ItOrExp,options] queries a human (ItOrExp takes a description of items) or a simulated expert (ItOrExp represents a simulated expert) on structure of the set Q of items. The established structure is returned either as a space or assertions. A valid option is QueryingAlgorithm (Koppen [default]). For additional valid options see QEKQueryExpertByKoppen.

QSRMakeSonde

QSRMakeSonde[OQ,NQ] returns a querying session observation sonde consisting of the number OQ of open questions at the start of the observation and the number NQ of questions presented to an expert during the observation.

QSRSondeOpenQuestionsInitial

QSRSondeOpenQuestionsInitial[Sonde] returns the initial number of open questions recorded by the querying session observation sonde Sonde.

QSRSondeOpenQuestionsInitialAdd

QSRSondeOpenQuestionsInitialAdd[Sonde,N] returns the querying session sonde Sonde after incrementing the initial number of open questions by N.

QSRSondeOpenQuestionsInitialReplace

QSRSondeOpenQuestionsInitialReplace[Sonde,N] returns the querying session sonde Sonde after replacing the initial number of open questions with N.

QSRSondeQuestionsAsked

QSRSondeQuestionsAsked[Sonde] returns the number of questions presented to an expert recorded by the querying session sonde Sonde.

QSRSondeQuestionsAskedAdd

QSRSondeQuestionsAskedAdd[Sonde,N] returns the querying session sonde Sonde after incrementing the number of presented questions by N.

QSRSondeQuestionsAskedReplace

QSRSondeQuestionsAskedReplace[Sonde,N] returns the querying session sonde Sonde after replacing the number of asked questions by N.

ReadItems

ReadItems[FN,"KSMP_ItemsFile"] reads the description of items stored in the file FN using the file format "KSMP_ItemsFile".

ReadItems[FN] assumes the file format "KSMP_ItemsFile" is used.

ReadSkills

ReadSkills[FN,"KSMP_SkillsFile"] reads the description of skills stored in the file FN using the file format "KSMP_SkillsFile".

ReadSkills[FN] assumes the file format "KSMP_SkillsFile" is used.

ReadStructure

ReadStructure[FN] reads the structure stored in the file FN using "KST_SpaceFile" as default datafile format.

ReadStructure[FN,"KST_SpaceFile"] reads the structure stored in the file FN using "KST_SpaceFile" datafile format.

ReadStructure[FN,SC] reads the structure stored in the file FN and returns it rewritten using the carrier set SC, assuming the default datafile format.

ReadStructure[FN,"KST_SpaceFile",SC] reads the structure (using KST_SpaceFile datafile format) stored in the file FN and rewrites it using the carrier set SC.

RelationAntiSymmetricQ

RelationAntiSymmetricQ[SR] returns True iff the relation SR is antisymmetric.

RelationAReflexiveQ

RelationAReflexiveQ[SR] returns True iff the relation SR is areflexive.

RelationChangeCarrier

RelationChangeCarrier[SR,S1] returns the relation SR after changing the original elements with the elements of the set S1.

RelationChangeCarriers

RelationChangeCarriers[SR,S11,S22] returns the relation SR after changing the original elements from the set S1 with the elements from the set S11, and the elements of S2 with the elements from S22 respectively.

RelationChangePairs

RelationChangePairs[SR,R1] is a primitive function that returns the relation based on the relation SR after replacing the set R of ordered pairs by the set R1 of ordered pairs.

RelationChangeSet

RelationChangeSet[SR,S1] is a primitive function that returns the relation based on the relation SR after replacing the set S by the set S1.

RelationChangeSets

RelationChangeSets[SR,S11,S22] is a primitive function that returns the relation based on the relation SR after replacing the set S1 by the set S11, and the set S2 by the set S22.

RelationComplement

RelationComplement[SR] returns the complement of the relation SR.

RelationConnectedQ

RelationConnectedQ[SR] returns True iff the relation SR is connected.

RelationEquivalenceQ

RelationEquivalenceQ[SR] returns True iff the relation SR is an equivalence relation.

RelationIdentity

RelationIdentity[S] returns the identity relation on the set S.

RelationLinearOrderQ

RelationLinearOrderQ[SR] returns True iff the relation SR is a linear order.

RelationMaximalQ

RelationMaximalQ[SR,a] returns True iff the element a is a maximal with respect to the relation SR .

RelationMinimalQ

RelationMinimalQ[SR,a] returns True iff the element a is a minimal with respect to the relation SR .

RelationPairs

RelationPairs[SR] returns the set R of ordered pairs forming the relation (S,R) or (S1,S2,R).

RelationPairsCardinality

RelationPairsCardinality[SR] returns the cardinality of the family of ordered pairs forming the relation SR.

RelationPartialOrderQ

RelationPartialOrderQ[SR] returns True if the relation SR is a partial order.

RelationQ

RelationQ[SR] returns True iff SR is a relation.

RelationQuasiOrderQ

RelationQuasiOrderQ[SR] returns True iff the relation SR is a quasi order.

RelationRan

RelationRan[SR,a] returns all elements from the set forming the relation SR that are in the relation with the element a.

RelationReflexiveQ

RelationReflexiveQ[SR] returns True iff the relation SR is reflexive.

RelationSet

RelationSet[SR] returns the set forming the relation SR.

RelationSetCardinality

RelationSetCardinality[SR] returns the cardinality of the set forming the relation SR.

RelationSets

RelationSets[SR] returns the data structure {S1,S2} consisting of the sets forming the relation SR from the set S1 to the set S2.

RelationSetsCardinality

RelationSetsCardinality[SR] returns the cardinalities of the sets forming the relation SR.

RelationSubsetProperQ

RelationSubsetProperQ[SR1,SR2] returns True iff the relation SR1 is a proper subset of the relation SR2.

RelationSubsetQ

RelationSubsetQ[SR1,SR2] returns True iff the relation SR1 is a subset of the relation SR2.

RelationSymmetricQ

RelationSymmetricQ[SR] returns True iff the relation SR is symmetric.

RelationTransitiveQ

RelationTransitiveQ[SR] returns True if the relation SR is transitive.

RowList2String

RowList2String[R] rewrites the list R of numbers as the string.

RowString2List

RowString2List[R] rewrites the string R of digits as the list of numbers.

SetCardinality

SetCardinality[S] returns the cardinality of the finite set S.

SetCartesianProduct

SetCartesianProduct[S1,S2] returns the cartesian product of the non-empty sets S1 and S2.

SetCartesianProduct[{S1,S2,...,Sk}] returns the cartesian product of the non-empty sets S1,S2,...,Sk.

SetComplement

SetComplement[U,S] returns the complement of the set S in the universe U.

SetDifference

SetDifference[A,B] returns the set difference A\B.

SetElementQ

SetElementQ[a,S] returns True iff a is an element of the set S.

SetEmptyQ

SetEmptyQ[S] returns True iff the set S is empty.

SetNonEmptyQ

SetNonEmptyQ[S] returns True iff the set S is non-empty.

SetQ

SetQ[L] returns True iff L represents a set.

SkillCode

SkillCode[Skill] returns the code of the given skill Skill.

SkillDescription

SkillDescription[Skill] returns the description of the given skill Skill.

SkillGet

SkillGet[Skills,Code] returns the first skill described in Skills whose code matches Code.

SkillName

SkillName[Skill] returns the name of the given skill Skill.

SSAttributionQ

SSAttributionQ[LofCl] returns True iff a non-empty collection of clauses is assigned to each item within the mapping LofCL of items to clauses.

SSAxiom1Q

SSAxiom1Q[LofCl] returns True iff the mapping LofCl of items to clauses is an attribution.

SSAxiom2atItemQ

SSAxiom2atItemQ[q,Clauses] returns True iff the item q is an element of all clauses that are assigned to it in the collection Clauses of clauses.

SSAxiom2Q

SSAxiom2Q[LofCl] returns True iff each item in the mapping LofCL is member of all clauses assigned to it.

SSAxiom2Q[Q,LofCl] returns True iff each item from Q is member of all clauses assigned to it by the mapping LofCL of items to clauses.

SSAxiom3Q

SSAxiom3Q[LofCl] returns True iff for each clause C of any item q given by the mapping LofCl, there exist a clause C1 assigned to each item q1 of the clause C, that is a subset of the clause C.

SSAxiom4atClausesQ

SSAxiom4atClausesQ[clauses] returns True iff any two clauses for the same item are incomparable.

SSAxiom4Q

SSAxiom4Q[LofCl] returns True iff any two clauses of any item, within the mapping LofCl, are incomparable.

SSClausesOfItem

SSClausesOfItem[SS,q] returns the clauses assigned to the item q by the surmise function forming the surmise system SS.

SSListOfClauses

SSListOfClauses[SS] returns the list of clauses of the surmise system SS.

SSLofClClausesGet

SSLofClClausesGet[LofCl,q] returns the clauses assigned to the item q by the mapping LofCl of items to clauses.

SSSFClausesOfItem

SSSFClausesOfItem[SF,q] returns the clauses assigned to the item q by the surmise function SF.

SSSFListOfClauses

SSSFListOfClauses[SF] returns the list of clauses of the surmise function SF.

SSSurmiseFunction

SSSurmiseFunction[SS] returns the surmise function forming the surmise system SS.

SSSurmiseSet

SSSurmiseSet[SS] returns the set forming the surmise system SS.

StateBinary2Set

StateBinary2Set[B,Normalized] returns the set representation of the state given its binary list representation B, under assumption that the structure to which the state will belong to is normalized.

StateBinary2Set[B,SC] returns the set representation of the state given its binary list representation B and the structure carrier set SC.

StateSet2Binary

StateSet2Binary[f,NS,Normalized] returns the binary representation of the state f, under assumption that the cardinality of the normalized structure carrying set is NS.

StateSet2Binary[f,SC] returns the binary representation of the state f taking into account the carrier set SC of the structure to which it belongs to.

StructureFamily

StructureFamily[SF] returns the family forming the structure SF.

StructureFamilyElementQ

StructureFamilyElementQ[SF,f] returns True iff the set f is a element of the family forming the structure SF.

StructureFamilyEmptyQ

StructureFamilyEmptyQ[SF] returns True iff the family forming the structure SF is empty.

StructureFamilyNonEmptyQ

StructureFamilyNonEmptyQ[SF] returns True iff the family forming the structure SF is non-empty.

StructureFamilySubsetQ

StructureFamilySubsetQ[SF,F1] returns True iff the family F1 is a subset of the family forming the structure SF.

StructureFromMatrix

StructureFromMatrix[M,Normalized] returns the normalized structure from its matrix representation M.

StructureFromMatrix[M,SC] returns the structure from its matrix representation having the set SC as the carrier set.

StructureNormalize

StructureNormalize[SF] returns the normalized structure SF, that is rewrites the structure SF using the set {1,2,..} for carrying the structure.

StructureNormalize[SF,S1] returns the rewritten structure SF using the set S1 for carrying the structure.

StructureQ

StructureQ[SF] returns True iff SF is a structure.

StructureReplaceCarrier

StructureReplaceCarrier[SF,SC] returns the rewritten structure SF using the set SC as the new carrier.

StructureReplaceFamily

StructureReplaceFamily[SF,F1] is a primitive function that returns the structure SF after replacing its family with the family F1.

StructureReplaceSet

StructureReplaceSet[SF,X1] is a primitive function that returns the structure SF after replacing its set with the set X1.

StructureSet

StructureSet[SF] returns the set forming the structure SF.

StructureSetElementQ

StructureSetElementQ[SF,s] returns True iff the element s is a member of the set forming the structure SF.

StructureSetEmpty

StructureSetEmptyQ[SF] returns True iff the set forming the structure SF is empty.

StructureSetNonEmptyQ

StructureSetNonEmptyQ[SF] returns True iff the set forming the structure SF is non-empty.

StructureSetSubsetQ

StructureSetSubsetQ[SF,S1] returns True iff the set S1 is a subset of the set forming the structure SF.

StructureSubsetProperQ

StructureSubsetProperQ[SF1,SF2] returns True iff the structure SF1 is a proper subset of the structure SF2. Both structures are based on the same set.

StructureSubsetQ

StructureSubsetQ[SF1,SF2] returns True iff the structure SF1 is contained in the structure SF2. Both structures are based on the same set.

StructureToMatrix

StructureToMatrix[SF] returns the matrix representation of the structure SF.

StructureToMatrix[SF,Normalized] returns the matrix representation of the structure SF, under assumption that the structure SF is normalized.

StudentAnswerCertaintyFactor

StudentAnswerCertaintyFactor[Ans] returns the certainty factor of the student's answer Ans.

StudentAnswerItem

StudentAnswerItem[Ans] returns the logical value of the student's answer Ans.

StudentAnswerLogicalValue

StudentAnswerLogicalValue[Ans] returns the logical value of the student's answer Ans.

StudentCarelessErrors

StudentCarelessErrors[S] returns the uncertainty that the student S makes a careless error.

StudentKnowledge

StudentKnowledge[S] returns domain knowledge of the student S.

StudentLuckyGuesses

StudentLuckyGuesses[S] returns the uncertainty that the student S makes a lucky guess.

StudentMakeAnswer

StudentMakeAnswer[QE,LV,CF] returns the data structure for a student's answer consisting of the asked question QE, logical value LC describing the correctness of the answer and the certainty factor CF describing the student's certainty in the given answer.

SubsetProperQ

SubsetProperQ[A,B] returns True iff the set A is a proper subset of the set B.

SubsetQ

SubsetQ[A,B] returns True iff the set A is a subset of the set B.

SurmiseFunctionFromBasis

SurmiseFunctionFromBasis[B] returns the surmise function calculated from the basis B.

SurmiseFunctionFromKnowledgeSpace

SurmiseFunctionFromKnowledgeSpace[QK] returns the surmise function calculated from the knowledge space QK.

SurmiseFunctionQ

SurmiseFunctionQ[f] returns True iff the function f represents a surmise function.

SurmiseSystemFromBasis

SurmiseSystemFromBasis[B] returns the surmise system calculated from the basis B.

SurmiseSystemFromKnowledgeSpace

SurmiseSystemFromKnowledgeSpace[QK] returns the surmise system representation of the given knowledge space QK.

SurmiseSystemQ

SurmiseSystemQ[QS] returns True iff the data structure QS represents a surmise system.

SurmiseSystemToBasis

SurmiseSystemToBasis[SS] returns the basis calculated from the surmise system SS.

SurmiseSystemToKnowledgeSpace

SurmiseSystemToKnowledgeSpace[SS] returns the basis calculated from the surmise system SS.

WarpQ

WarpQ[X] returns True iff X is the warp representation of a list of positive integers.

WriteItems

WriteItems[FN,Items] stores the description on the items Items into the file FN.

WriteSkills

WriteSkills[FN,Skills] stores the description on the skills Skills into the file FN.

WriteStructure

WriteStructure[FN,SF,"KST_SpaceFile"] writes the structure SF in the file FN using "KST_SpaceFile" datafile format.

WriteStructure[FN,SF] assumes "KST_SpaceFile" as the datafile format.

KSMP source code

version 0.91 (beta 1)

KSMP warp core

```
(* :Title: Knowledge Spaces Mathematica Package, Warp Core.  *)

(* :Context: `KnowledgeSpaces`WarpCore` *)

(* :Author: Andrej Zaluski *)

(* :Summary:
        KSMP Warp Core. Provides data structures to the layer one that are
       suitable for environments with limited memory.
 *)

(*  :Copyright: 20001, Andrej Zaluski.
        See attached license.
 *)

(*  :Package Version: 0.91 beta 1 *)

(*  :Mathematica Version: 4.0 *)

(*  :History:
           1.0 the first public release.
           0.1 first draft. *)

(*  :Keywords:  *)

(*  :Sources: *)

(*  :Warnings: BETA 1 - TESTING PHASE *)

(*  :Limitations: *)

(*  :Discussion: *)

(*  :Requirements:
                   NONE *)




BeginPackage["KnowledgeSpaces`WarpCore`"]



WarpQ::usage="WarpQ[X] returns True iff X is the warp representation of a \
list of positive integers.";

KSMPListToBinary::usage="KSMPListToBinary[L]returns the binary \
representation list for the list L consisting of positive integers.";

KSMPBinaryToList::usage="KSMPBinaryToList[BL]returns the list of positive \
integers calculated from the binary list BL.";

KSMPListToWarp::usage="KSMPListToWarp[L]returns the warp representation of \
the list L of positive integers.";

KSMPBinaryToWarp::usage="KSMPBinaryToWarp[BL]returns the warp representation \
```

```
of the binary representation list BL representing a list of positive \
integers."

KSMPWarpToList::usage="KSMPWarpToList[W]returns the list of positive \
integers from its warp representation W.";

KSMPWarpToBinary::usage="KSMPWarpToBinary[W]returns the binary list \
representation of the given warp W."



Begin["`Private`"]



WarpQ[W_]:=Return[IntegerQ[W]]

KSMPListToBinary[{}]={};
KSMPListToBinary[l_List]:=Module[{max,b,i},
    max=Max[l];
    b=Table[0,{max}];
    Do[b[[(max+1-l[[i]])]]=1;,{i,1,Length[l]}];
    Return[b];
    ]/; Apply[And,Map[(IntegerQ[#]&&Positive[#])&,l]]

KSMPBinaryToList[{}]={};
KSMPBinaryToList[BL_List]:=Module[{n=Length[BL],i,L={}},
    Do[If[BL[[i]]==1,AppendTo[L,(n+1-i)]];,{i,n,1,-1}];
    Return[L]
    ] /; Apply[And,Map[((#==1||#==0))&,BL]]

KSMPBinaryToWarp[BL_List]:=Module[{},
    Return[FromDigits[BL,2]]
    ] /; Apply[And,Map[((#==1||#==0))&,BL]]

KSMPListToWarp[{}]=0;
KSMPListToWarp[L_List]:=Module[{},
    Return[FromDigits[KSMPListToBinary[L],2]]
    ] /; Apply[And,Map[(IntegerQ[#]&&Positive[#])&,L]]

KSMPWarpToList[0]={};
KSMPWarpToList[W_?WarpQ]:=Module[{},
  KSMPBinaryToList[IntegerDigits[W,2]]
  ];

KSMPWarpToBinary[W_?WarpQ]:=IntegerDigits[W,2];




End[ ]


EndPackage[ ]

(* Print["KSMP v1.0 Warp Core ... OK."]; *)
Print["Knowledge Spaces Mathematica Package."];
Print["(C) 2001 Andrej Zaluski. Free software (see attached license)."];
```

KSMP layer one

```
(* :Title: Knowledge Spaces Mathematica Package - Layer One. *)

(* :Context: KnowledgeSpaces`L1` *)

(* :Author: Andrej Zaluski *)

(* :Summary:
The package KnowledgeSpaces`L1` represents the abstract data types layer one \
of the KnowledgeSpaces` Mathematica Package KSMP.The aim of this package is \
to be an interface between basic Mathematica data structures and functions at \
one side,and Mathematica implemented procedures representing concepts and \
procedures from the knowledge space theory on the other.Data structures and \
functions from this abstract data types layer of the package are to be used \
by other layers of the KnowledgeSpaces` package.The layer consists of the \
components representing sets,relations,functions,
  structures (here defined as ordered tuples consisting of sets and families \
of subsets),
  a component with functions representing miscellaneous operations and \
property tests,
  and finally a component dealing with data input and output of results and \
reports.The reading and writing of data files is done in a way compatible \
with already developed Unix (C/C++,
      Bash) tools for dealing with knowledge spaces (primarily,
       KST tools written by Cord Hockemeyer).
  *)

(*  :Copyright: 20001, Andrej Zaluski.
          See attached license.
  *)

(*  :Package Version: 0.9 beta 1 *)

(*  :Mathematica Version: 4.0 *)

(*  :History:
            1.0 reserved for the first public version.
            0.9 beta 1 version. *)

(*  :Keywords:
     Psychology and Education; Mathematical Psychology;
  Knowledge Modeling; Knowledge Space Theory
  *)

(*  :Sources: *)

(*  :Warnings: *)

(*  :Limitations:
     Currently,  on personal computers large part  of writen functions works well only for \
small sets of items. Future plans include devloping two operating modes of the package: \
'research mode' and 'application mode'. Research mode will be suitable for research and teaching, \
explorations, and simulations on small sets of items,
  while the application mode will be able to perform all the tasks on sets of \
items of relevant size for real-life applications.
  *)

(*  :Discussion: *)
```

```
(*  :Requirements:
                Packages: DiscreteMath`Combinatorica`, KnowledgeSpaces`WarpCore`
 *)



BeginPackage["KnowledgeSpaces`L1`","DiscreteMath`Combinatorica`","KnowledgeSpaces`WarpCore`"]



(* TODO *) (* Save the state of the package *)
Off[General::spell,General::spell1]



SetQ::usage="SetQ[L] returns True iff L represents a set."

SetCardinality::usage="SetCardinality[S] returns the cardinality of the \
finite set S."

SetNonEmptyQ::usage="SetNonEmptyQ[S] returns True iff the set S is non-empty."

SetEmptyQ::usage="SetEmptyQ[S] returns True iff the set S is empty."

SubsetQ::usage="SubsetQ[A,B] returns True iff the set A is a  subset of the \
set B."

EqualSetsQ::usage="EqualSetsQ[A,B] returns True iff the sets A and B are \
equal."

SubsetProperQ::usage="SubsetProperQ[A,B] returns True iff the set A is a \
proper subset of the set B."

SetElementQ::usage="SetElementQ[a,S] returns True iff a is an element of the \
set S."

SetCartesianProduct::usage="SetCartesianProduct[S1,S2] returns the Cartesian \
Product of the non-empty sets S1 and S2. SetCartesianProduct[{S1,S2,...}] \
returns the Cartesian product of the non-empty sets S1,S2,... "

PowerSet::usage="PowerSet[S] returns the powerset of the set S. By default \
ranking, all subsets with the cardinality k-1 appear before the subsets with \
the cardinality k."

SetDifference::usage="SetDifference[A,B] returns the set difference A\B."

SetComplement::usage="SetComplement[U,S] returns the complement of the set S \
in the universe U."



MakeRelation::usage="MakeRelation[S,R] returns a data structure representing \
the relation given by the set S and the subset R of the cartesian product S\
×S.
\n\nMakeRelation[S1,S2,R] returns a data structure representing the relation \
```

from the set S1 to the set S2 given by the subset R of the cartesian product \
S1×S2."

RelationQ::usage="RelationQ[SR] returns True iff SR is a relation."

RelationSet::usage="RelationSet[SR] returns the set forming the relation SR."

RelationSets::usage="RelationSets[SR] returns the data structure {S1,S2} \
consisting of the sets forming the relation (S1,S2,R)."

RelationPairs::usage="RelationPairs[SR] returns the set R of ordered pairs \
forming the relation (S,R) or (S1,S2,R)."

RelationChangeSet::usage=" RelationChangeSet[SR,S1] returns the relation \
based on the relation SR in which the set S was replaced by the set S1."

RelationChangeSets::usage="RelationChangeSets[SR,S11,S22] is a primitive \
function that returns the relation based on the relation SR after replacing \
the set S1 by the set S11, and the set S2 by the set S22."

RelationChangePairs::usage="RelationChangePairs[SR,R1] is a primitive \
function that returns the relation based on the relation SR after replacing \
the set R of ordered pairs by the set R1 of ordered pairs."

RelationSetCardinality::usage="RelationSetCardinality[SR] returns the \
cardinality of the set forming the relation SR."

RelationSetsCardinality::usage="RelationSetsCardinality[SR] returns the \
cardinalities of the sets forming the relation R."

RelationPairsCardinality::usage="RelationPairsCardinality[SR] returns the \
cardinality of the family of ordered pairs forming the relation SR."

RelationChangeCarrier::usage="RelationChangeCarrier[SR,S1] returns the \
relation SR after changing the original elements with the elements from the \
set S1."

RelationChangeCarriers::usage="RelationChangeCarriers[SR,S11,S22] returns the \
relation SR after changing the original elements from the set S1 with the \
elements from the set S11, and the elements from S2 with the elements from \
S22 respectively."

ElementsInRelationQ::usage="ElementsInRelationQ[SR,a,b] returns True iff the \
elements a and b are in the relation SR."

RelationMaximalQ::usage="RelationMaximalQ[SR,\[Alpha]] returns True iff the \
element \[Alpha] is a maximal with respect to the relation SR ."

RelationMinimalQ::usage="RelationMinimalQ[SR,\[Alpha]] returns True iff the \
element \[Alpha] is a minimal with respect to the relation SR ."

MaximalsOfRelation::usage="MaximalsOfRelation[SR] returns all maximals of the \
relation SR."

MinimalsOfRelation::usage="MinimalsOfRelation[SR] returns all minimals of the \
relation SR."

RelationComplement::usage="RelationComplement[SR] returns the complement of \
the relation SR."

```
RelationIdentity::usage="RelationIdentity[S] returns the identity relation on \
the set S."

EqualRelationsQ::usage="EqualRelationsQ[SR1,SR2] returns True iff the \
relations SR1 and SR2 are equal."

RelationSubsetQ::usage="RelationSubsetQ[SR1,SR2] returns True iff the \
relation SR1 is a subset of the relation SR2."

RelationSubsetProperQ::usage="RelationSubsetProperQ[SR1,SR2] returns True iff \
the relation SR1 is a proper subset of the relation SR2."



MakeFunction::usage="MakeFunction[D,C,M] creates a function from the set D to \
the set C using the mapping rule M."

FunctionQ::usage="FunctionQ[F] returns True iff F represents a function."

FunctionDomain::usage="FunctionDomain[f] returns the domain of the function \
f."

FunctionCodomain::usage="FunctionCodomain[f] returns the codomain of the \
function f."

FunctionMapping::usage="FunctionMapping[f] returns the mapping rule of the \
function f."

FunctionChangeDomain::usage="FunctionChangeDomain[f,D1] is a primitive \
function that returns the function f after replacing its domain with the new \
set D1."

FunctionChangeCodomain::usage="FunctionChangeCodomain[f,C1] is a primitive \
function that returns the function f after replacing its codomain with the \
new set C1."

FunctionChangeMapping::usage="FunctionChangeMapping[f,M1] is a primitive \
function that returns the function f after replacing its mapping rule with \
the new mapping rule M1."

FunctionMappingOriginal::usage="FunctionMappingOriginal[f] returns
the function's original calculated using the mapping rule of the function f."

FunctionMappingImage::usage="FunctionMappingImage[f] returns the function's \
image calculated using the mapping rule of the function f."

FunctionWellDefinedQ::usage="FunctionWellDefinedQ[f] returns True iff the \
original of a mapping rule of the function f is equal to the domain of the \
function."

FunctionSurjectionQ::usage="FunctionSurjectionQ[f] returns True iff the \
function f is a surjection."

FunctionValue::usage="
FunctionValue[f,x] returns the function value f(x) of the element x.
\n\nFunctionValue[functions,x] returns the value of x after applying the list \
of
functions (from right to left).
\n\nFunctionValue[f,D1] returns  values of the elements from the set D1 after \
applying the function f.
```

```
\n\nFunctionValue[functions,D1] returns values of the elements from the set \
D1 after applying the list of functions (from right to left)."

FunctionSubsetImage::usage="FunctionSubsetImage[f,D1] returns the image of \
the subset D1 under the function f.
\n\nFunctionSubsetImage[functions,D1] returns the image of the subset D1 \
after applying a list of functions (from right to left)."

FunctionInjectionQ::usage="FunctionInjectionQ[f] returns True iff the \
function f is an injection."

FunctionBijectionQ::usage="FunctionBijectionQ[f] returns True iff the \
function f is a bijection."

FunctionRestriction::usage="FunctionRestriction[f,D1] returns the function f \
restricted to its subdomain D1."

CompositionOfFunctions::usage="CompositionOfFunctions[{fn,..,f1}] return the \
composition
(fn o .. o f1) of the functions fn,..,f1 under assumption that
such composition is defined."

EqualFunctionsQ::usage="EqualFunctionsQ[f1,f2] returns True iff the functions \
f1 and f2
  are equal."



MakeStructure::usage="MakeStructure[X,F] returns a data structure \
representing the structure consisting of the set X of items and the family F \
of subsets from the set X."

StructureQ::usage="StructureQ[SF] returns True iff SF is a structure."

FamilyQ::usage="FamilyQ[F] returns True iff SF is a family of subsets."

StructureSet::usage="StructureSet[SF] returns the set forming the structure \
SF."

StructureFamily::usage="StructureFamily[SF] returns the family forming the \
structure SF."

StructureReplaceSet::usage="StructureReplaceSet[SF,X1] is a primitive \
function that returns the structure SF after replacing the set forming the \
structure by the set X1."

StructureReplaceFamily::usage="StructureReplaceFamily[SF,F1] is a primitive \
function that returns the structure SF after replacing its family by the \
family F1."

StructureReplaceCarrier::usage="StructureReplaceCarrier[SF,SC] returns the \
rewritten structure SF using the set SC as the new carrier."

StructureSetEmpty::usage="StructureSetEmptyQ[SF] returns True iff the set \
forming the structure SF is empty."

StructureSetNonEmptyQ::usage="StructureSetNonEmptyQ[SF] returns True iff the \
set forming the structure SF is non-empty."

StructureFamilyEmptyQ::usage="StructureFamilyEmptyQ[SF] returns True iff the \
```

```
family forming the structure SF is empty."

StructureFamilyNonEmptyQ::usage="StructureFamilyNonEmptyQ[SF] returns True \
iff the family forming the structure SF is non-empty."

StructureSetElementQ::usage="StructureSetElementQ[SF,s] returns True iff the \
element s is an element of the set forming the structure SF."

StructureSetSubsetQ::usage="StructureSetSubsetQ[SF,S1] returns True iff the \
set S1 is a subset of the set forming the structure SF."

StructureFamilyElementQ::usage="StructureFamilyElementQ[SF,f] returns True \
iff the set f is an element of the family forming the structure SF."

StructureFamilySubsetQ::usage="StructureFamilySubsetQ[SF,F1] returns True iff \
the family F1 is a subset of the family forming the structure SF."

StructureSubsetQ::usage="StructureSubsetQ[SF1,SF2] returns True iff the \
structure SF1 is contained in the structure SF2. Both structures are based on \
the same set."

EqualStructuresQ::usage="EqualStructuresQ[SF1,SF2] returns True iff the \
structure SF1 is equal to the structure SF2."

StructureSubsetProperQ::usage="StructureSubsetProperQ[SF1,SF2] returns True \
iff the structure SF1 is a proper subset of the structure SF2. Both \
structures are based on the same set."

StateBinary2Set::usage="StateBinary2Set[B,Normalized] returns the set \
representation of the state given its binary list representation B, under \
assumption that the structure to which the state will belong to is \
normalized.\n\n
StateBinary2Set[B,SC] returns the set representation of the state given its \
binary list representation B and the structure carrier set SC."

StateSet2Binary::usage="StateSet2Binary[f,NS,Normalized] returns the binary \
representation of the state f, under assumption that the cardinality of the \
normalized structure carrying set S is NS.
StateSet2Binary[f,SC] returns the binary representation of the state f
  taking into account the carrier set SC of the structure to which it belongs \
to."

StructureNormalize::usage="StructureNormalize[SF] returns the normalized \
structure SF, i.e. rewrites the structure SF  taking the set {1,2,..} as the \
set carrying the structure.\n\n
StructureNormalize[SF,S1] returns the rewritten structure SF taking into \
account S1 as the set carrying the structure."

StructureFromMatrix::usage="StructureFromMatrix[M] returns the normalized \
structure from its matrix representation M.\n\n
StructureFromMatrix[M,Normalized] returns the normalized structure from its \
matrix representation M.\n\n
StructureFromMatrix[M,SC] returns the structure from its matrix \
representatition having the set SC as its carrier set."

StructureToMatrix::usage="StructureToMatrix[SF] returns the matrix \
representation of the structure SF.\n\n
StructureToMatrix[SF,Normalized] returns the matrix representation of the \
structure SF assuming that the structure is already normalized."
```

```
ReadStructure::usage="ReadStructure[FN] reads the structure stored in the \
file FN taking \"KST_SpaceFile\" as default data file format.\n\n
ReadStructure[FN,\"KST_SpaceFile\"] reads the structure stored in the file FN \
using the data file format \"KST_SpaceFile\".\n\n
ReadStructure[FN,SC] reads the structure stored in the file FN and rewrites \
it using the carrier set SC, assuming the default datafile format.\n\n
ReadStructure[FN,\"KST_SpaceFile\",SC] reads the structure (using \
KST_SpaceFile datafile format) stored in the file FN and rewrites it using \
the carrier set SC."

WriteStructure::usage="WriteStructure[FN,SF,\"KST_SpaceFile\"]writes the \
structure SF in the
file FN using \"KST_SpaceFile\" datafile format.\n\n
WriteStructure[FN,SF] assumes \"KST_SpaceFile\" as the datafile format."

FX::usage="FX[SF,s] returns all members of the family forming the structure \
SF, which contain the element s of the set forming the structure.
\n\nFX[SF,C,NC] returns all members of the family forming the structure SF, \
which contain all the elements from the set C and have empty intersectio with \
the set NC.
\n\nFX[SF,C] returns all members of the family forming the structure SF, \
which contain all the elements from the set C."



DataFileRead::usage="DataFileRead[Filename] creates a datafile object \
representing the data file Filename."

DataFileLength::usage="DataFileLength[SFile] returns the number of lines in \
the datafile object SFile."

LineCommentQ::usage="LineCommentQ[L] returns True iff the line L of a \
datafile object contains a comment."

CommentLines::usage="CommentLines[SFile] returns the list of line numbers \
from the datafile object SFile that contain comments."

NonCommentLines::usage="NonCommentLines[SFile]returns the list of all line \
numbers of the datafile object that do not contain a comment."

DataFileCommentExtract::usage="DataFileCommentExtract[SFile] returns the list \
representing comments from the datafile object SFile."

DataFileCommentPrint::usage="DataFileCommentPrint[DF] extracts and prints the \
comments from the datafile object DF."

DataFileCommentDrop::usage="DataFileCommentDrop[SFile] returns the list \
representing the datafile object SFile after removing comments."

FileFormatsInfo::usage="FileFormatsInfo[FileFormat,Info]returns the \
information Info about the requested file format FileFormat. Currently, \
supported formats are: \"KST_SpaceFile\", \"KST_PatternFile\". The stored \
information about the formats is: \"Head\"."

DataFileHeadExtract::usage="DataFileHeadExtract[SFL,FileFormat] returns the \
extracted head from the non-commented data list SFL representing the datafile \
object in accordance with the datafile format FileFormat. \n\nFor the \
supported datafile formats see FileFormatsInfo."

DataFileBodyExtract::usage="DataFileBodyExtract[SFL,FileFormat] returns the \
```

```
extracted  body from the non-commented data list SFL representing the \
datafile object using the file format FilFormat.\n\nFor the supported \
datafile formats see FileFormatsInfo."

DataFileHeadInterpret::usage="DataFileHeadInterpret[H,FF]interprets the \
extracted head H from a datafile object in the FF format."

DataFileBodyInterpret::usage="DataFileBodyInterpret[B,FF]interprets the \
extracted body B from the data file object of the FF format."

RowList2String::usage="RowList2String[R] rewrites the list R of numbers as \
the string."

RowString2List::usage="RowString2List[R] rewrites the string R of digits as \
the list of numbers."

Matrix2DataFileBody::usage="Matrix2DataFileBody[M_List,\"KST_SpaceFile\"]\
returns the datafile object body from the matrix M using KST_SpaceFile \
format."

Matrix2DataFileHead::usage="Matrix2DataFileHead[M,\"KST_SpaceFile\"] returns \
the head of a datafile object from the matrix M usingKST_SpaceFile format."

DataFileWrite::usage="DataFileWrite[FN,M,\"KST_SpaceFile\"]writes the matrix \
M, together with the derived head, in the datafile FN using the KST_SpaceFile \
format.
\n\nDataFileWrite[FN,Items,\"KSMP_ItemsFile\"]writes the description Items \
into the file FN using the format KSMP_ItemsFile.
\n\nDataFileWrite[FN,Skills,\"KSMP_SkillsFile\"]writes the description \
Skills into the file FN using the format KSMP_SkillsFile."




ForAllQ::usage="ForAllQ[X,pf]returns True iff the property pf holds for all \
elements of the set X."

ExistOneQ::usage="ExistOneQ[X,pf] returns True iff the property pf holds for \
at least one element of the set X."

ForAllPairsQ::usage="ForAllPairsQ[X,pf]returns True iff the predicate pf \
gives True for all ordered pairs from X.
\n\nForAllPairsQ[X,Y,pf] returns True iff the predicate pf gives True for all \
pairs from X x Y."

ForAllTriplesQ::usage="ForAllTriplesQ[X,pf]returns True iff the predicate pf \
gives True for all ordered triples from X.
ForAllTriplesQ[X,Y,Z,pf] returns True iff the predicate pf returns True for \
all ordered triples from X x Y x Z."



$Verbosity::usage="$Verbosity is the default of the option Verbosity."

$Verbosity=0  (* global default *)

Verbosity::usage="Verbosity is an option of many functions from the upper \
layers and modules of the KSMP package. The admissible values are Off \
(default), numbers (e.g. 1,5,10,15).";
```

```
(* FORMAT Relation[S,R],OutputForm *)
Format[SR:Relation[S_,R_],OutputForm]:=Module[{},
   "("<>ToString[S]<>","<>
    ToString[Map[(If[#=={},\[EmptySet],
          "("<>ToString[#[[1]]]<>","<>ToString[#[[2]]]<>")"])&,
       R]]<>")"
   ]

(* FORMAT Structure[S,F], OutputForm *)
Format[SF:Structure[S_,F_],OutputForm]:=Module[{S1,F1},
   S1:=S /. {}->\[EmptySet];
   F1:=F /.{}->\[EmptySet];
   ("("<>ToString[S1]<>","<>ToString[F1]<>")")
   ]


Begin["`Private`"]




(* SetQ[L] returns True if L is a set *)
SetQ[L_]:=If[Head[L]===List,True,False]

(* SetCardinality[S] returns the cardinal number of the finite set S *)
SetCardinality[S_List]:=Length[Union[S]]

(* SetNonEmptyQ[S] returs true if the set S is non-empty *)
SetNonEmptyQ[S_]:=If[Length[S]\[NotEqual]0,True,False]
(* SetEmptyQ[S] returns true if the set S is empty *)
SetEmptyQ[S_]:=If[Length[S]==0,True,False]

(* SubsetQ[A,B] tests whether the set A is a subset of the set B *)
SubsetQ[A0_List,B0_List]:=Module[{A=A0,B=B0},
   A=Union[A0];
   B=Union[B0];
   Which[A==={},True,
         B==={},False,
         True,  (Intersection[A,B]\[Equal]A)

    ]
   ]

(* EqualSetsQ[A,B] tests whethther the sets A and B are equal *)
EqualSetsQ[A_List,B_List]:=SubsetQ[A,B]\[And]SubsetQ[B,A]

(* SubsetProperQ[A,
    B] tests whether the set A is a proper subset of the set B *)
SubsetProperQ[A0_List,B0_List]:=SubsetQ[A0,B0]\[And]A0\[NotEqual]B0

(* SetElementQ[a,S] tests whether a is an element of the set S *)
SetElementQ[a0_,S0_List]:=Module[{a=a0,S=S0},
   Which[
    a==={},SubsetQ[{{}},S],
    True, SubsetQ[List[a],S]
    ]
   ]
```

```
(* SetNonEmptyQ[S] returs true if the set S is non-empty *)
SetNonEmptyQ[S_]:=If[Length[S]\[NotEqual]0,True,False]
(* SetEmptyQ[S] returns true if the set S is empty *)
SetEmptyQ[S_]:=If[Length[S]==0,True,False]

(* SetCartesianProduct[S1,S2] returns the Cartesian Product of the non-
   empty sets S1 and S2 *)
(* SetCartesianProduct[{S1,
      S2,...}] returns the Cartesian product of the sets S1,S2,... *)
SetCartesianProduct[S1_List?SetNonEmptyQ,S2_List?SetNonEmptyQ]:=
 CartesianProduct[S1,S2]
SetCartesianProduct[L_List]:=Module[{N=Length[L],tmp},
  If[N===2,
    Return[SetCartesianProduct[L[[1]],L[[2]]]];,
    tmp:=
      Map[(Flatten[#,1])&,
        SetCartesianProduct[First[L],
          Map[(Flatten[#,1])&,SetCartesianProduct[Rest[L]]]]];
    Return[tmp];
    ]
  ]

(* PowerSet[
    X] returns the powerset of the set X with the ranking where all the \
subsets with the size (k-1) appear before the subsets of the size k*)
PowerSet[X_]:=Module[{i},
  Apply[Join,Table[KSubsets[X,i],{i,0,Length[X]}]]
  ]

(* SetDifference[A,B] returns the set difference A\B *)
SetDifference[A_,B_]:=Complement[A,B] /; SetQ[A]\[And]SetQ[B]

(* SetComplement[U,S] returns the complement of the set S in the universe U *)


SetComplement[U_,S_]:=SetDifference[U,S] /; SubsetQ[S,U]



SetQ[X_]:=If[Head[X]===List || Head[X]===Integer,True,False]

SetEmptyQ[S_?WarpQ]:=If[S==0,True,False]

SetNonEmptyQ[S_?WarpQ]:=Not[SetEmptyQ[S]]

SetCardinality[S_?WarpQ]:=If[S!=0,Apply[Plus,KSMPWarpToBinary[S]],0];

SetIntersection[A_?WarpQ,B_?WarpQ]:=BitAnd[A,B]

SetUnion[A_?WarpQ,B_?WarpQ]:=BitOr[A,B]

SubsetQ[W1_?WarpQ,W2_?WarpQ]:=SetIntersection[W1,W2]===W1;

EqualSetsQ[W1_?WarpQ,W2_?WarpQ]:=W1===W2;

SubsetProperQ[A_?WarpQ,B_?WarpQ]:=SubsetQ[A,B]&&Not[EqualSetsQ[A,B]];

SetElementQ[e_Integer?Positive,B_?WarpQ]:=SubsetQ[KSMPListToWarp[{e}],B]
```

```
SetDifference[A_?WarpQ,B_?WarpQ]:=Module[
   {BLA=KSMPWarpToBinary[A],BLB=KSMPWarpToBinary[B],nA,nB,n,i},
   nA=Length[BLA];nB=Length[BLB];
   n=Max[{nA,nB}];
   Do[PrependTo[BLA,0],{i,nA,n-1}];
   Do[PrependTo[BLB,0],{i,nB,n-1}];
   KSMPBinaryToWarp[((BLA-BLB)/. -1->0)]
   ]

SetComplement[U_?WarpQ,A_?WarpQ]:=SetDifference[U,A]



(* MakeRelation[S,
    R] returns the data object Relation for the given set S and the sebset \
R of the Cartesian product SxS *)
MakeRelation[Set_List,Pairs_List]:=Relation[Set,Pairs]

(* MakeRelation[S1,S2,
    R] returns the data object representing the relation from the set S1 to \
the set S2 given by the subset R of the Cartesian Product S1xS2. *)
MakeRelation[S1_List,S2_List,Pairs_List]:=Relation[S1,S2,Pairs]

(* RelationQ[SR] returns True is SR is a relation *)
RelationQ[Relation[S_,R_]]:=SubsetQ[R,SetCartesianProduct[S,S]]
RelationQ[Relation[S1_,S2_,R_]]:=SubsetQ[R,SetCartesianProduct[S1,S2]]

(* Defining primitive functions *)
(* RelationSet[SR] returns S from the relation (S,R) *)
RelationSet[Relation[S_,R_]]:=S

(* RelationSets[SR]returns the sets {S1,S2} forming the relation (S1,S2,R)*)
RelationSets[Relation[S1_,S2_,R_]]:={S1,S2}

(* RelationPairs[SR] returns the pairs R from the relation (S,R) or (S1,S2,
    R) *)
RelationPairs[Relation[S_,R_]]:=R
RelationPairs[Relation[S1_,S2_,R_]]:=R

(* HIDDEN: Within (S,R), replaces S by S1 *)
RelationChangeSet[Relation[S_,R_],S1_]:=Module[{SR=Relation[S,R]},
   Relation[S1,RelationPairs[SR]]
   ]

(* HIDDEN: Wihtin (S1,S2,R) replaces S1 by S11, S2 by S22 *)
RelationChangeSets[Relation[S1_,S2_,R_],S11_,S22_]:=
  Module[{SR=Relation[S1,S2,R]},
   Relation[S11,S22,RelationPairs[SR]]
   ]

(* HIDDEN: Within (S,R) replaces R by R1 *)
RelationChangePairs[Relation[S_,R_],R1_]:=Module[{SR:=Relation[S,R]},
   Relation[RelationSet[SR],R1]
   ]

(* HIDDEN: Within (S1,S2,R) replaces R by R1 *)
RelationChangePairs[Relation[SA_,SB_,R_],R1_]:=
  Module[{SR:=Relation[SA,SB,R]},
   Relation[RelationSets[SR][[1]],RelationSets[SR][[2]],R1]
```

```
   ]


(* RelationSetCardinality[
     SR] returns the cardinality of the set forming the relation SR *)
RelationSetCardinality[Relation[S_,R_]]:=Module[{SR:=Relation[S,R]},
   SetCardinality[RelationSet[SR]]
   ]

(* RelationSetsCardinality[
     SR] returns the cardinalities of the sets forming the relation R from \
the set S1 to the set S2 *)
RelationSetsCardinality[Relation[SA_,SB_,R_]]:=
  Module[{SR:=Relation[SA,SB,R],RelSets},
    RelSets:=RelationSets[SR];
    {SetCardinality[RelSets[[1]]],SetCardinality[RelSets[[2]]]}
    ]

(* RelationPairsCardinality[
     SR] returns the cardinality of the familiy of ordered pairs forming the \
relation SR  *)
RelationPairsCardinality[SR_Relation]:=SetCardinality[RelationPairs[SR]]
  (* Works for both (S,R) and (S1,S2,R) *)

(* RelationChangeCarrier[SR,
     Items] rewrites the relation SR by changing relation's set with the set \
Items
      labels of the items  *)
 RelationChangeCarrier[Relation[S0_,R0_],Items0_List]:=
  Module[{SR0:=Relation[S0,R0],Items:=Items0,S,R,Changes,SR},
     SR:=SR0;
     Items=Union[Items0];
     S=RelationSet[SR];
     R=RelationPairs[SR];
     Changes=Table[S[[i]]->Items[[i]],{i,1,Length[S]}];
     S= (S /. Changes);
     R= (R /. Changes);
     SR=RelationChangeSet[SR,S];
     SR=RelationChangePairs[SR,R];
     Return[SR]
     ] /; (Length[Items0]===Length[S0] &&
             Length[Items0]===SetCardinality[Items0])
    (* The changing mapping is a bijective function *)


(* RelationChangeCarriers[SR,Items1,
     Items2] rewrites the relation SR by changing relation's first set with \
the set Items1, and the second set respectively *)
RelationChangeCarriers[Relation[S01_,S02_,R0_],Items01_List,Items02_List]:=
  Module[{Items1,Items2,S1,S2, R,Changes1,Changes2,tmp1,tmp2},
     Items1:=Union[Items01];Items2:=Union[Items02];
     S1:=S01;S2:=S02;R:=R0;tmp1:={};tmp2:={};
      Changes1=Table[S1[[i]]->Items1[[i]],{i,1,Length[S1]}];
     Changes2=Table[S2[[i]]->Items2[[i]],{i,1,Length[S2]}];
     S1= (S1 /. Changes1);
     S2= (S2 /. Changes2);
     tmp1=((Transpose[R][[1]]) /. Changes1);
     tmp2=((Transpose[R][[2]]) /. Changes2);
     R:=Transpose[{tmp1,tmp2}];
     Return[Relation[S1,S2,R]]
```

```
    ] /; (Length[Items01]===Length[S01] &&
      Length[Items02]===Length[S02] &&
            Length[Items01]===SetCardinality[Items01] &&
      Length[Items02]===SetCardinality[Items02])
  (* The changing mappings are bijective functions *)


(* ElementsInRelationQ[SR,a,
    b] tests whether elements a and b are in the relation SR *)
ElementsInRelationQ[Relation[S_,R_],a_,b_] :=Module[{},
    SetElementQ[{a,b},R]
    ] /; SubsetQ[{a,b},S]
ElementsInRelationQ[Relation[S1_,S2_,R_],a_,b_] :=Module[{},
    SetElementQ[{a,b},R]
    ] /; SubsetQ[{a},S1]\[And]SubsetQ[{b},S2]

(* RelationMaximalQ[
    SR,\[Alpha]] tests whether the element \[Alpha] is the maximal with \
respect to the relation SR *)
RelationMaximalQ[Relation[S_,R_],\[Alpha]_]:=Module[{SR:=Relation[S,R]},
    Apply[And,
      Map[(ElementsInRelationQ[SR,\[Alpha],#]\[Implies]\[Alpha]\[Equal]#)&,
        RelationSet[SR]]]
    ]/; SetElementQ[\[Alpha],S]



(* RelationMinimalQ[
    SR,\[Alpha]] tests whether the element \[Alpha] is the minimal with \
respect to the relation SR *)
RelationMinimalQ[Relation[S_,R_],\[Alpha]_]:=Module[{SR:=Relation[S,R]},
    Apply[And,
      Map[(ElementsInRelationQ[SR,#,\[Alpha]]\[Implies]\[Alpha]\[Equal]#)&,
        RelationSet[SR]]]
    ]/; SetElementQ[\[Alpha],S]


(* MaximalsOfRelation[SR] returns all the maximals for the relation SR *)
MaximalsOfRelation[Relation[S_,R_]]:=Module[{SR:=Relation[S,R]},
  DeleteCases[Map[(If[RelationMaximalQ[SR,#1],#1])&,S],Null]
  ]


(* MinimalsOfRelation[SR] returns all the minimals of the relation SR *)
MinimalsOfRelation[Relation[S_,R_]]:=Module[{SR:=Relation[S,R]},
  DeleteCases[Map[(If[RelationMinimalQ[SR,#1],#1])&,S],Null]
  ]

(* RelationComplement[SR] returns the complement of the relation SR *)
RelationComplement[Relation[S_,R_]]:=Module[{},
  Return[Relation[S,SetComplement[SetCartesianProduct[S,S],R]]]
  ]
RelationComplement[Relation[S1_,S2_,R_]]:=Module[{},
  Return[Relation[S1,S2,SetComplement[SetCartesianProduct[S1,S2],R]]]
  ]

(* RelationIdentity[S] returns the identity relation on the set S *)
RelationIdentity[S_]:=Return[Relation[S,Map[({#,#})&,S]]]
```

```
(* EqualRelationsQ[SR1,
    SR2] returns true iff the relations SR1 and SR2 are equal *)
EqualRelationsQ[Relation[S1_,R1_],Relation[S2_,R2_]]:=
 EqualSetsQ[S1,S2]\[And]EqualSetsQ[R1,R2]
EqualRelationsQ[Relation[S11_,S12_,R1_],Relation[S21_,S22_,R2_]]:=
 EqualSetsQ[S11,S21]\[And]EqualSetsQ[S12,S22]\[And]EqualSetsQ[R1,R2]


(* RelationSubsetQ[SR1,
    SR2] returns true iff the relation SR1 is a subset of relation SR2 *)
RelationSubsetQ[Relation[S1_,R1_],Relation[S2_,R2_]]:=
 EqualSetsQ[S1,S2]\[And]SubsetQ[R1,R2]
RelationSubsetQ[Relation[S11_,S12_,R1_],Relation[S21_,S22_,R2_]]:=
 EqualSetsQ[S11,S21]\[And]EqualSetsQ[S12,S22]\[And]SubsetQ[R1,R2]

(* RelationSubsetProperQ[SR1,
    SR2] returns true iff the relation SR1 is a proper subset of relation \
SR2 *)
RelationSubsetProperQ[Relation[SA_,RA_],Relation[SB_,RB_]]:=
 Module[{SRA:=Relation[SA,RA],SRB:=Relation[SB,RB]},
  RelationSubsetQ[SRA,SRB]\[And]¬EqualRelationsQ[SRA,SRB]
  ]
RelationSubsetProperQ[Relation[SA1_,SA2_,RA_],Relation[SB1_,SB2_,RB_]]:=
 Module[{SRA:=Relation[SA1,SA2,RA],SRB:=Relation[SB1,SB2,RB]},
  RelationSubsetQ[SRA,SRB]\[And]¬EqualRelationsQ[SRA,SRB]
  ]



(* MakeFunction[D,C,
    M] creates a function from the set D to the set C using the mapping M *)


MakeFunction[D_List,C_List,M_List]:=Mapping[D,C,M] /; Length[D]===Length[M]

(* FunctionQ[F] returns True if F is a data structure containing a function *)


FunctionQ[F_]:=If[Head[F]===Mapping,True,False]

(* FunctionDomain[f] returns the domain of the function f *)
FunctionDomain[Mapping[d_,c_,m_]]:=d

(* FunctionCodomain[f] returns the codomain of the function f*)
FunctionCodomain[Mapping[D_,C_,M_]]:=C

(* FunctionMapping[f] returns the mapping rule of the function f*)
FunctionMapping[Mapping[D_,C_,M_]]:=M

(* FunctionChangeDomain[f,
    D1] replaces the domain of the function f with the new set D1 *)
FunctionChangeDomain[F_Mapping,D1_]:=
 Mapping[D1,FunctionCodomain[F],FunctionMapping[F]]

(* FunctionChangeCodomain[f,
    C1] replaces the domain of the function f with the new set C1 *)
FunctionChangeCodomain[F_Mapping,C1_]:=
 Mapping[FunctionDomain[F],C1,FunctionMapping[F]]

(* FunctionChangeMapping[f,
```

```
    M1] replaces the mapping rule of the function f with the new mapping \
rule M1 *)
FunctionChangeMapping[F_Mapping,M1_]:=
 Mapping[FunctionDomain[F],FunctionCodomain[F],M1]

(* FunctionMappingOriginal[
    f] returns the original of the function's mapping rule*)
FunctionMappingOriginal[F_Mapping]:=Transpose[FunctionMapping[F]][[1]]
(* FunctionMappingImage[f] returns the function's image *)
FunctionMappingImage[F_Mapping]:=Union[Transpose[FunctionMapping[F]][[2]]]

(* FunctionWellDefinedQ[
    f] returns true if the original of the function's mapping rule is equal \
to the domain of the function *)
FunctionWellDefinedQ[f_Mapping]:=
 EqualSetsQ[FunctionDomain[f],FunctionMappingOriginal[f]]

(* FunctionSurjectionQ[f] returns true if the function f is a surjection *)
FunctionSurjectionQ[f_Mapping]:=
 EqualSetsQ[FunctionMappingImage[f],FunctionCodomain[f]]

(* FunctionValue[f,x] returns the f(x) *)
FunctionValue[Mapping[D0_,C0_,M0_],x_]:=Module[{M=M0},
    Select[M,(#[[1]]===x)&,1][[1,2]]
    ]  /; SetElementQ[x,FunctionMappingOriginal[Mapping[D0,C0,M0]]]

(* FunctionValue[functions,
    x] applies a list of functions (from right to left) to value of x  *)
FunctionValue[{f_Mapping},x_]:=FunctionValue[f,x]
FunctionValue[functions_List,x_]:=FunctionValue[First[functions],
  FunctionValue[Rest[functions],x]]

(* FunctionValue[f,
    D1] returns a list of function values of the elements of the set D1  *)


FunctionValue[f_Mapping,D0_List]:=Module[{D=D0},
    Map[(FunctionValue[f,#])&,D]
    ] /; SubsetQ[D0,FunctionDomain[f]]

\!\(\*
 RowBox[{
  RowBox[{"(*", " ",
    RowBox[{\(FunctionValue[functions, D1]\), " ", "applies", " ", "a", " ",
      "list", " ", "of", " ", "functions",
      " ", \((from\ right\ to\ the\ left)\), " ", "on", " ", "the", " ",
      "subset", " ", "of", " ", "the", " ", "first", " ", "domain", " ",
      D1}], " ", "*)"}],
  "\[IndentingNewLine]", \(FunctionValue[functions_List, D0_List] :=
    Module[{D =
        D0}, \[IndentingNewLine]Map[\((FunctionValue[
          functions, #])\) &, D]\[IndentingNewLine]]\  /; \
     SubsetQ[D0,
      FunctionDomain[\(Take[functions, \(-1\)]\)[\([\)\(1\)\(]\)]]]\)}]\)

(* FunctionSubsetImage[f,
    D1] returns the image of the subset D1 under the function f *)
FunctionSubsetImage[f_Mapping,D0_List]:=Module[{D=Union[D0]},
    Union[Map[(FunctionValue[f,#])&,D]]
    ] /; SubsetQ[D0,FunctionDomain[f]]
```

```
(* FunctionInjectionQ[f] tests whether the function f is an injection *)
FunctionInjectionQ[Mapping[D0_,C0_,M0_]]:=
 Module[{D=D0,C=C0,M=M0,DxD=CartesianProduct[D0,D0],f=Mapping[D0,C0,M0]},
    Apply[And,
     Map[(#[[1,1]]\[NotEqual]#[[1,2]]\[Implies]#[[2,1]]\[NotEqual]#[[2,
           2]])&,
      Table[{DxD[[i]],
        {FunctionValue[f,DxD[[i,1]]],FunctionValue[f,DxD[[i,2]]]}},
        {i,1,Length[DxD]}
       ]]]] /; FunctionWellDefinedQ[Mapping[D0,C0,M0]]




(* CompositionOfFunctions[
    functions]returns the composition(fn\[SmallCircle]..\[SmallCircle]
     f2\[SmallCircle]f1) of the functions (taking from right to left),
 in case that such composition is defined *)
CompositionOfFunctions[functions_List]:=Module[
    {i,k=Length[functions],A1,f},
     A1=FunctionDomain[functions[[k]]];
     f=Mapping[A1,
               FunctionCodomain[functions[[1]]],
               Transpose[{A1,FunctionValue[functions,A1]}]];
    Return[f]
    ] /; (Length[functions]\[GreaterEqual]2) \[And]
        (Apply[And,Map[(Head[#]===Mapping)&,functions]]) \[And]
         (Apply[And,Table[SubsetQ[FunctionCodomain[functions[[i+1]]],
                                                  FunctionDomain[functions[[i]]]],
                                          {i,Length[functions]-1,1,-1}]])

(* EqualFunctionsQ[f1,f2] returns true iff the functions f1 and f2
   are equal *)
EqualFunctionsQ[Mapping[D1_,C1_,R1_],Mapping[D2_,C2_,R2_]]:=
 EqualSetsQ[D1,D2]&&EqualSetsQ[C1,C2]&&EqualSetsQ[R1,R2]



(* MakeStructure[X,F] creates a structure (X,F) *)
MakeStructure[X_List,F_List]:=Structure[X,F]


(* StructureQ[SF] returns True if SF is a structure *)
StructureQ[SF_]:=If[Head[SF]===Structure,True,False]

(* FamilyQ[F] returns True if SF is a family of subsets *)
FamilyQ[F_]:=SetQ[F]

(* StructureSet[SF] returns the set forming the structure SF *)
StructureSet[Structure[X_,F_]]:=X

(* StructureFamily[SF] returns the family forming the structure SF *)
StructureFamily[Structure[X_,F_]]:=F

(* StructureReplaceSet[SF,
    X1] primitive function that replaces the set forming the structure SF \
```

```
with the set X1 *)
StructureReplaceSet[Structure[X_,F_],X1_]:=Structure[X1,F]

(* StructureReplaceFamily[SF,
    F1] primitive function that replaces the family forming the structure \
with the family F1*)
StructureReplaceFamily[Structure[X_,F_],F1_]:=Structure[X,F1]



(* StructureSetEmptyQ[
    SF] returns True if the set forming the structure SF is empty *)
StructureSetEmptyQ[Structure[X_,F_]]:=SetEmptyQ[X]

(* StructureSetNonEmptyQ[
     SF] returns True if the set forming the structure SF is non-empty *)
StructureSetNonEmptyQ[Structure[X_,F_]]:=SetNonEmptyQ[X]

(* StructureFamilyEmptyQ[
    SF] returns True if the family forming the structure SF is empty*)
StructureFamilyEmptyQ[Structure[X_,F_]]:=SetEmptyQ[F]

(* StructureFamilyNonEmptyQ[
     SF] returns True if the family forming the structure SF is non-empty *)


StructureFamilyNonEmptyQ[Structure[X_,F_]]:=SetNonEmptyQ[F]


(* StructureSetElementQ[SF,
    s] tests whether s is a element of the set forming the structure SF *)
StructureSetElementQ[Structure[S_,F_],s_]:=SetElementQ[s,S]

(* StructureSetSubsetQ[SF,
    S1] tests whether the set S1 is a subset of the set forming the \
structure SF *)
StructureSetSubsetQ[Structure[S_,F_],S1_]:=SubsetQ[S1,S]

(* StructureFamilyElementQ[SF,
    f] tests whether the set f is a element of the family forming the \
structure SF *)
StructureFamilyElementQ[Structure[S_,F_],f_]:=SetElementQ[f,F]

(* StructureFamilySubsetQ[SF,
    F1] tests whether the family F1 is a subset of the family forming the \
structure SF *)
StructureFamilySubsetQ[Structure[S_,F_],F1_]:=SubsetQ[F1,F]

(* StructureSubsetQ[SF1,
    SF2] tests whether the structure SF1 is contained (in the sense S1=S2,
    F1\[SubsetEqual]F2), in the structure SF2*)
StructureSubsetQ[Structure[S1_,F1_],Structure[S2_,F2_]]:=Module[{},
    If[SubsetQ[F1,F2],True,False]
    ] /; EqualSetsQ[S1,S2]

(* EqualStructuresQ[SF1,
    SF2] tests whether the structure SF1 is equal to the structure SF2 *)
EqualStructuresQ[Structure[S1_,F1_],Structure[S2_,F2_]]:=Module[{},
    EqualSetsQ[F1,F2]
    ] /; EqualSetsQ[S1,S2]
```

```
(* StructureSubsetProperQ[SF1,
     SF2]tests whether the structure SF1 is a proper subset (in the sense S1=
      S2, F1\[SubsetEqual]F2) of the structure SF2 *)
StructureSubsetProperQ[SF1_Structure,SF2_Structure]:=
 StructureSubsetQ[SF1,SF2]\[And] ¬EqualStructuresQ[SF1,SF2]

(* StateBinary2Set[B,
     Normalized] returns the set representation of the state given its \
binary list representation B,
 under assumption that the structure to which the state will belong to is \
normalized  *)
(* StateBinary2Set[B,
     S1] returns the set representation of the state given its binary list \
representation B and the structure carrier set S1 *)
StateBinary2Set[B_List,Normalized]:=Flatten[Position[B,1]]
StateBinary2Set[B_List,S1_List]:=Module[{},
     Return[
      StateBinary2Set[B,Normalized] /.
       Map[(#[[1]]->#[[2]])&,Transpose[{Range[Length[B]],S1}]]
      ]
     ] /; Length[B]===Length[S1]

(* StateSet2Binary[f,NS,
     Normalized] returns the binary representation of the state f,
 under assumption that the normalized structure carrying set S contains NS \
elements *)
(* StateSet2Binary[f,SC] returns the binary representation of the state f,
 taking into account the carrier set SC of the structure to which it belongs \
to *)
StateSet2Binary[f_List,NS_Integer,Normalized]:=Module[{tmp,ZeroRow},
   ZeroRow:=Table[0,{NS}];
   tmp=ZeroRow;tmp[[f]]=1;tmp
   ]
StateSet2Binary[f_List,S_List]:=Module[{NS=SetCardinality[S]},
     StateSet2Binary[
      f /. Map[(#[[1]]->#[[2]])&, Transpose[{S,Range[NS]}]],
      NS,Normalized]
     ]/; SubsetQ[f,S]

(* StructureNormalize[SF] rewrites ("normalizes") the structure SF,
 taking the set {1,2,3,..} as the set carrying the structure. *)
(* StructureNormalize[SF,
     S1] rewrites the structure SF taking into account S1 as the set \
carrying the structure *)
StructureNormalize[SR_Structure]:=
 StructureNormalize[SR,Range[SetCardinality[StructureSet[SR]]]]
StructureNormalize[Structure[S_,R_],S1_List]:=Module[{SR=Structure[S,R]},
     Return[SR/. Map[(#[[1]]->#[[2]])&,Transpose[{S,S1}]] ]
     ]/;SetCardinality[S]===SetCardinality[S1]

(* StructureFromMatrix[
     M] returns the normalized structure from its matrix representation M*)
(* StructureFromMatrix[M,
     Normalized] returns the normalized structure from its matrix \
representation M*)
(* StructureFromMatrix[M,
     SC]returns the structure from its matrix representatition having the \
set SC as the carrier set *)
StructureFromMatrix[M_List,SC_List]:=Module[{SF},
```

```
    SF=Structure[SC,
      (Map[(StateBinary2Set[#,Normalized])&,M] /.
        Map[(#[[1]]->#[[2]])&,Transpose[{Range[Length[M[[1]]]],SC}]])
      ];
    Return[SF]
    ] /; Length[M[[1]]]===Length[SC]
StructureFromMatrix[M_List,Normalized]:=
 StructureFromMatrix[M,Range[Length[M[[1]]]]]
StructureFromMatrix[M_List]:=StructureFromMatrix[M,Normalized]

(* StructureToMatrix[
    SF] returns the matrix representation of a structure SF *)
(* StructureToMatrix[SF,
    Normalized] returns the matrix representation of the structure,
 SF under assumption that the structure is already normalized *)
StructureToMatrix[SF_Structure]:=
 StructureToMatrix[StructureNormalize[SF],Normalized]
StructureToMatrix[Structure[S_,F_],Normalized]:=
 Module[{tmp,ZeroRow,CS=SetCardinality[S],i},
  ZeroRow:=Table[0,{CS}];
  Table[tmp=ZeroRow;tmp[[F[[i]]]]=1;tmp
   ,{i,1,Length[F]}
   ]
  ]


(* ReadStructure[
    FN] reads the structure stored in the file FN taking "KST_SpaceFile" as \
default data file format *)
(* ReadStructure[FN,
    "KST_SpaceFile"] reads the structure stored in the file FN taking \
"KST_SpaceFile" as the data file format *)
(* ReadStructure[FN,
    SC] reads the structure stored in the file FN and rewrites it using the \
carrier set SC *)
(* ReadStructure[FN,"KST_SpaceFile",
    SC] reads the structure stored in the file FN and rewrites it using the \
carrier set SC *)
ReadStructure[FN_String]:=ReadStructure[FN,"KST_SpaceFile",Normalized]
ReadStructure[FN_String,SC_List]:=ReadStructure[FN,"KST_SpaceFile",SC]
ReadStructure[FN_String,"KST_SpaceFile",Normalized]:=
 StructureFromMatrix[
  DataFileBodyInterpret[
   DataFileBodyExtract[DataFileCommentDrop[DataFileRead[FN]],
    "KST_SpaceFile"],"KST_SpaceFile"],
  Normalized]
ReadStructure[FN_String,"KST_SpaceFile",SC_List]:=
 StructureFromMatrix[
  DataFileBodyInterpret[
   DataFileBodyExtract[DataFileCommentDrop[DataFileRead[FN]],
    "KST_SpaceFile"],"KST_SpaceFile"],
  SC]

(* WriteStructure[FN,SF] writes the structure SF in the
  file FN with "KST_SpaceFile" as default datafile format *)
(* WriteStructure[FN,SF,"KST_SpaceFile"] writes the structure SF in the
  file FN according to "KST_SpaceFile" datafile format *)
WriteStructure[Filename_String,SF_Structure]:=
 WriteStructure[Filename,SF,"KST_SpaceFile"]
WriteStructure[Filename_String,SF_Structure,"KST_SpaceFile"]:=
```

```
 DataFileWrite[Filename,StructureToMatrix[SF],"KST_SpaceFile"]

(* FX[SF,s] returns the subfamily of the family F such that its elements \
contains s \[Element] S *)
FX[SF0_Structure,s0_]:=
 Module[{SF=SF0,s=s0},FX[SF,{s},{}]] /; Not[(Head[s0]===List)]
(* FX[SF,C,
      NC] returns the subfamily of the family SF such that its elements \
contain the C \[SubsetEqual]
   S and do not contain neither one of the elements NC \[SubsetEqual] S *)
FX[Structure[S_,F_],C_List,NC_List]:=Module[{},
    Select[
      Select[F,(SubsetQ[C,#])&],
      (Intersection[NC,#]==={})&
      ]
    ]/; Apply[And,
      Map[(SubsetQ[{#},S])&,Union[C,NC]]] \[And] (Intersection[C,NC]==={})
(* FX[SF,C] returns the subfamily of the family SF such that its elements \
contain the C \[SubsetEqual] S *)
FX[SF0_Structure,C0_List]:=Module[{SF=SF0,C=C0},FX[SF,C,{}]]



(* DataFileRead[
    Filename] creates the DataFile object representing the file Filename \
containing the data *)
DataFileRead[FileName_String]:=DataFile[Import[FileName,"Lines"]]

(* DataFileLength[
    SFile] returns the number of lines within the datafile object SFile *)
DataFileLength[DataFile[SFile_]]:=Length[SFile]

(* LineCommentQ[
    L] tests whether the line L of a datafile object contains a comment *)
LineCommentQ[Line_]:=LineCommentQ[Line,"KST_SpaceFile"]
LineCommentQ[Line_,FileFormat_]:=StringTake[Line,{1}]===
   FileFormatsInfo[FileFormat,"CommentSign"]

(* CommentLines[SFile] returns a list of line-
   numbers of a datafile object that contain comments *)
CommentLines[DataFile[SFile_]]:=
 Flatten[Position[Map[(LineCommentQ[#])&,SFile],True]]

(* NonCommentLines[SFile]returns a list of line-
   numbers of the datafile object that do not contain a comment *)
NonCommentLines[SFile_DataFile]:=
 Complement[Range[DataFileLength[SFile]],CommentLines[SFile]]

(* DataFileCommentExtract[
    SFile] returns a list representing the comments within the datafile \
object Sfile *)
DataFileCommentExtract[DataFile[SFileList_]]:=
 SFileList[[CommentLines[DataFile[SFileList]]]]

(* DataFileCommentPrint[
    DF] extracts and prints the comment from a datafile object DF *)
DataFileCommentPrint[DF_DataFile ]:=DataFileCommentExtract[DF] //TableForm

(* DataFileCommentDrop[
    SFile] returns the list representing the datafile object SFile with the \
```

```
comments removed *)
DataFileCommentDrop[DataFile[SFileList_]]:=
  Return[SFileList[[NonCommentLines[DataFile[SFileList]]]]]

(* FileFormatsInfo[FileFormat,
     Info] returns the information Info about the requested file format \
FileFormat. Currently,
  supported formats are: "KST_SpaceFile", "KST_PatternFile",
  "KSMP_ItemsFile" and "KSMP_SkillsFile".
     The stored information Info about the formats are:
     "CommentSign" and "Head". *)
FileFormatsInfo[FileFormat_,InfoType0_]:=Module[
   {FileFormatData={{"KST_SpaceFile","#",{1,2}},{"KST_PatternFile",
         "#",{1,2}},
       {"KSMP_ItemsFile","#",{}},{"KSMP_SkillsFile","#",{}}},
     InfoTypes={{"CommentSign",2},{"Head",3}},InfoType},
   InfoType:=Select[InfoTypes,(#[[1]]===InfoType0)&,1][[1,2]];
   Return[Select[FileFormatData,(#[[1]]===FileFormat)&,1][[1,InfoType]]]
   ]

(* DataFileHeadExtract[SFL,FileFormat] extracts the head from non-
   commented data list SFL representing the datafile object in accordance \
with the datafile format FileFormat.
      For the supported file formats see FileFormatsInfo[] *)
DataFileHeadExtract[SFL_,DataFileFormat_String]:=Module[{},
   Return[SFL[[FileFormatsInfo[DataFileFormat,"Head"]]]]
   ]

(* DataFileBodyExtract[SFL,FileFormat] extracts the body from non-
   commented data list SFL representing the datafile object,
  in accordance with the file format FilFormat.
     For the supported file formats see FileFormatsInfo[] *)
DataFileBodyExtract[SFL_,DataFileFormat_String]:=Module[{},
   SFL[[
     Complement[Range[Length[SFL]],FileFormatsInfo[DataFileFormat,"Head"]]
     ]]
   ]

(* DataFileHeadInterpret[H,
     FF] interprets the extracted head H from the data file object of the FF \
format *)
DataFileHeadInterpret[H_List,FF_String]:=Module[{},
   Switch[FF,
     "KST_SpaceFile",DFHI[H,"KST_SpaceFile"],
     "KST_PatternFile",DFHI[H,"KST_SpaceFile"]
     ]
   ]

(* Hidden - converts a list of strings to the list of numbers *)
DFHI[H_,"KST_SpaceFile"]:=Map[(ToExpression[#])&,H]

(* DataFileBodyInterpret[B,
     FF] interprets the extracted body B from the data file object of the FF \
format *)
DataFileBodyInterpret[B_List,FF_String]:=Module[{},
   Switch[FF,
     "KST_SpaceFile",DFBI[B,"KST_SpaceFile"],
     "KST_PatternFile",DFBI[B,"KST_SpaceFile"]
     ]
   ]
```

```
(* HIDDEN:
   Rewrites the body of the KST_Space file data file object to a matrix of \
numbers *)
DFBI[B_,"KST_SpaceFile"]:=Module[{j},
   Table[
    RowString2List[B[[j]]]
    ,{j,1,Length[B]}]
   ]

(* RowList2String[R] rewrites a list R of numbers as a string*)
RowList2String[Row_List]:=Module[{},StringJoin[Map[(ToString[#])&,Row]]]

(* RowString2List[R] rewrites a string of 1-digits as the list of numbers *)
RowString2List[Row_String]:=
 Module[{i},
   Table[ToExpression[StringTake[Row,{i,i}]],{i,1,StringLength[Row]}]]

(* Matrix2DataFileBody[M_List,
     "KST_SpaceFile"] creates the datafile object body,
 according to KST_SpaceFile format, from a matrix M *)
Matrix2DataFileBody[M_List,"KST_SpaceFile"]:=Module[{i},
   Table[RowList2String[M[[i]]],{i,1,Length[M]}]
   ]

(* Matrix2DataFileHead[M,
     "KST_SpaceFile"] produces the head part of datafile object,
 according to KST_SpaceFile format, from a matrix M *)
Matrix2DataFileHead[M_List,"KST_SpaceFile"]:=Module[{},
   Map[(ToString[#])&,{Length[M[[1]]],Length[M]}]
   ]

(* DataFileWrite[FN,M,"KST_SpaceFile"] writes a matrix M,
 together with the derived head,
 in the datafile FN respecting the KST_SpaceFile format *)
DataFileWrite[FileName_String,M_List,"KST_SpaceFile"]:=Module[{},
   Export[FileName,
    Join[
     Matrix2DataFileHead[M,"KST_SpaceFile"],
     Matrix2DataFileBody[M,"KST_SpaceFile"]
     ],
    "Lines"]
   ]

(* DataFileWrite[FN,Items,
     "KSMP_ItemsFile"] writes the Items information into the file FN using \
the format KSMP_ItemsFile *)
DataFileWrite[FileName_String,Items0_List,"KSMP_ItemsFile"]:=Module[{Items},
   (* Changes the 1st integer or symbol of each item to STRING *)
   Items=Map[(Join[{ToString[#[[1]]]},Rest[#]])&,Items0];
   Export[FileName,
    Join[{"# Items File"},
     {"# KSMP v1.0"},Flatten[Items,1]]
    ,"Lines"]
   ]

(* DataFileWrite[FN,Skills,
     "KSMP_SkillsFile"] writes the Skills information into the file FN using \
the format KSMP_SkillsFile *)
DataFileWrite[FileName_String,Skills0_List,"KSMP_SkillsFile"]:=
```

```
 Module[{Skills},
   (* Changes the 1st integer or symbol of each skill to STRING *)
   Skills=Map[(Join[{ToString[#[[1]]]},Rest[#]])&,Skills0];
   Export[FileName,
    Join[{"# Skills File"},
      {"# KSMP v1.0"},Flatten[Skills,1]]
    ,"Lines"]
   ]




(* ForAllQ[X,
    pf] returns True iff the property pf holds for all elements of the set \
X *)
ForAllQ[X_List,p_Function]:= Apply[And, Map[p,X]]

(* ExistOneQ[X,
    pf] returns True iff the property pf holds for at least one element of \
the set X *)
ExistOneQ[X_List,p_Function]:=Apply[Or,Map[p,X]]

(* ForAllPairsQ[X,
    pf] returns True iff the predicate pf gives True for all ordered pairs \
from X *)
ForAllPairsQ[X_List?SetQ,pf_Function]:=ForAllPairsQ[X,X,pf]
(* ForAllPairsQ[X,Y,
    pf] returns True iff the predicate pf gives True for all pairs from X x \
Y *)
ForAllPairsQ[X_List?SetQ,Y_List?SetQ,pf_Function]:=
 ForAllQ[SetCartesianProduct[X,Y],pf]

(* ForAllTriplesQ[X,
    pf] returns True iff the predicate pf gives True for all the ordered \
triples from X *)
ForAllTriplesQ[X_List?SetQ,pf_Function]:=ForAllTriplesQ[X,X,X,pf]

(* ForAllTriplesQ[X,Y,Z,
    pf] returns True iff the predicate pf returns True for all the ordered \
triples from X x Y x Z *)
ForAllTriplesQ[X_List?SetQ,Y_List?SetQ,Z_List?SetQ,pf_Function]:=Module[{},
   ForAllQ[SetCartesianProduct[{X,Y,Z}],pf]
   ]



End[]


EndPackage[]
```

KSMP layer two

```
(* :Title: Knowledge Spaces Mathematica Package - Layer Two. *)

(* :Context: KnowledgeSpaces`L2` *)

(* :Author: Andrej Zaluski *)

(* :Summary:
        See KSMP technical report.
 *)

(*  :Copyright: 20001, Andrej Zaluski.
        See attached license.
 *)

(*  :Package Version: 0.9 beta 1 *)

(*  :Mathematica Version: 4.0 *)

(*  :History:
          1.0 reserved for the first public version.
          0.9 beta 1 version. *)

(*  :Keywords:
    Psychology and Education; Mathematical Psychology;
 Knowledge Modeling; Knowledge Space Theory
 *)

(*  :Sources: *)

(*  :Warnings: *)

(*  :Limitations:
        See Limitations of KSMP layer 1.
 *)

(*  :Discussion: *)

(*  :Requirements:
                Packages: DiscreteMath`Combinatorica` ,
 KnowledgeSpaces`Skills` (Modeling Experts Component)
 *)


(* TO DO *) (* Add also KSMP_Skills.m? *)
BeginPackage["KnowledgeSpaces`L2`",{"KnowledgeSpaces`L1`","KnowledgeSpaces`Skills`"}]

(* TODO *) (* Save the state of the package *)
Off[General::spell,General::spell1]


MakeItem::usage="MakeItem[Code,Desc,QText,CorrAns,LofDist]returns a data \
object representing the item consisting of the unique identifier Code, \
description Desc, text QText of the question, correct Answer CorrAns, and the \
list LofDist of admissible answers, where each answer consists of a code and \
a textual description."

ItemCode::usage="ItemCode[Item]returns the code of the given Item."
```

```
ItemDescription::usage="ItemDescription[Item] returns the description of the \
given Item."

ItemText::usage="ItemText[Item]returns the textual question of the given \
Item."

ItemCorrectAnswer::usage="ItemCorrectAnswer[Item]returns the correct answer \
of the given Item."

ItemAnswers::usage="ItemAnswers[Item]returns the list of all admissible \
answers on the item Item, where each answer consists of a code and a \
description."

ItemGet::usage="ItemGet[Items,code]returns the first item in the description \
whose code matches the code Code."

ReadItems::usage="ReadItems[FN,\"KSMP_ItemsFile\"]reads the description of \
items stored in the file FN using
 the file format \"KSMP_ItemsFile\".\n\n
ReadItems[FN] assumes the file format \"KSMP_ItemsFile\".";

WriteItems::usage="WriteItems[FN,Items]stores the description on the items \
Items into the file FN."

MakeSkill::usage="MakeSkill[Code,Name,Desc] returns the data structure \
representing a skill consisting of the unique identifier Code, the name Name \
and the description Desc."

SkillCode::usage="SkillCode[Skill] returns the code of the given skill Skill."

SkillName::usage="SkillName[Skill] returns the name of the given skill Skill."

SkillDescription::usage="SkillDescription[Skill] returns the description of \
the given Skill."

SkillGet::usage="SkillGet[Skills,Code] returns the first skill described in \
Skills whose code matches Code."

ReadSkills::usage="ReadSkills[FN,\"KSMP_SkillsFile\"] reads the description \
of skills stored in the file FN using the file format \"KSMP_SkillsFile\".\n\n\

ReadSkills[FN] assumes the file format \"KSMP_SkillsFile\"."

WriteSkills::usage="WriteSkills[FN,Skills] stores the description on the \
skills Skills into the file FN."


MakeExpert::usage="MakeExpert[MK,CE] returns the data structure representing \
an expert consisting of domain's metaknowledge MK and the uncertainty CE of \
making a careless error during a querying session."

ExpertMetaKnowledge::usage="ExpertMetaKnowledge[E]returns a domain \
metaknowledge of the expert E."

ExpertCarelessErrors::usage="ExpertCarelessErrors[E] returns the uncertainty \
that the expert E makes a careless error during a querying session."

MakeMetaKnowledge::usage="MakeMetaKnowledge[SF,Structure] returns a data \
structure representing experts knowledge in the form of structure for the \
```

given knowledge structure SF."

MakeMetaKnowledge::usage="MakeMetaKnowledge[PAss,NAss,UAss,Assertions]\
returns a data structure representing expert's metaknowledge in the form of \
assertions, for separately given lists of positive assertions PAss, negative \
assertions Nass and unknown assertions UAss."

MakeMetaKnowledge::usage="MakeMetaKnowledge[Ent,Assertions]returns a data \
structure representing expert's metaknowledge in the form of assertions for \
the given Entailment Ent."

MetaKnowledgeAssertionsPositive::usage="MetaKnowledgeAssertionsPositive[\
Assertions] returns all positive assertions contained in the expert's \
metaknowledge Assertions."

MetaKnowledgeAssertionsNegative::usage="MetaKnowledgeAssertionsNegative[\
Assertions] returns all negative assertions contained in the expert's \
metaknowledge Assertions."

MetaKnowledgeAssertionsRest::usage="MetaKnowledgeAssertionsRest[Assertions]\
returns all neither positive nor negative assertions contained in the \
expert's metaknowledge Assertions."

MakeMetaKnowledge::usage="MakeMetaKnowledge[DIAG,Diagnostic]returns a data \
structure representing an expert's metaknowledge, in terms of the \
competence-performance approach, using the diagnostic Diag."

MakeSimpleExpert::usage="MakeSimpleExpert[SF] returns an ideal expert \
possesing domains metaknowledge given by knowledge space SF."

MakeSimpleExpert::usage="MakeSimpleExpert[Diag] returns an ideal expert whose \
domain metaknowledge is given by the competence-performance diagnostic Diag."

ExpertMakeAnswer::usage="ExpertMakeAnswer[LV,CF]returns the data structure \
for an expert's answer consisting of the logical value LC and the certainty \
factor CF."

ExpertAnswerLogicalValue::usage="ExpertAnswerLogicalValue[Ans]returns the \
logical value of the expert's answer Ans."

ExpertAnswerCertaintyFactor::usage="ExpertAnswerCertaintyFactor[Ans]returns \
the certainty factor of the expert's answer Ans."

MakeStudent::usage="MakeStudent[QK,CE,LG] returns the data structure \
representing a student whose domain knowledge is given by the set QK of items \
the student is capable of solving, the uncertainty CE of making a careless \
error, and the uncertainty LG of making a lucky guess during a knowledge \
assessment."

StudentKnowledge::usage="StudentKnowledge[Stud] returns the domain knowledge \
of the student Stud."

StudentCarelessErrors::usage="StudentCarelessErrors[S] returns the \
uncertainty that the student S makes a careless error."

StudentLuckyGuesses::usage="StudentLuckyGuesses[S] returns the uncertainty \
that the student S makes a lucky guess."

```
MakeKnowledge::usage="MakeKnowledge[QK,Items]returns a data structure \
representing a student's domain knowledge given as the set QK of items."

MakeSimpleStudent::usage="MakeSimpleStudent[QK] returns an ideal student \
whose domain knowledge is given by the set QK of items."

StudentMakeAnswer::usage="StudentMakeAnswer[QE,LV,CF] returns the data \
structure for a student's answer consisting of the asked question QE, logical \
value LC describing the correctness of the answer and the certainty factor CF \
describing the student's certainty in the given answer."

StudentAnswerItem::usage="StudentAnswerItem[Ans]returns the answered item \
within the student's answer Ans."

StudentAnswerLogicalValue::usage="StudentAnswerLogicalValue[Ans]returns the \
logical value of the student's answer Ans."

StudentAnswerCertaintyFactor::usage="StudentAnswerCertaintyFactor[Ans]\
returns the certainty factor of the student's answer Ans."



RelationReflexiveQ::usage="RelationReflexiveQ[SR] returns True iff the \
relation SR is reflexive."

RelationAReflexiveQ::usage="RelationAReflexiveQ[SR]returns True iff the \
relation SR is areflexive."

RelationSymmetricQ::usage="RelationSymmetricQ[SR] returns True iff the \
relation SR is symmetric."

RelationAntiSymmetricQ::usage="RelationAntiSymmetricQ[SR] returns True iff \
the relation SR is antisymmetric."

RelationTransitiveQ::usage="RelationTransitiveQ[SR] returns True iff the \
relation SR is transitive."

RelationConnectedQ::usage="RelationConnectedQ[SR] returns True iff the \
relation SR is connected."

RelationEquivalenceQ::usage="RelationEquivalenceQ[SR] returns True iff the \
relation SR is an equivalence relation."

RelationQuasiOrderQ::usage="RelationQuasiOrderQ[SR] returns True if the \
relation SR is a quasi order."

RelationPartialOrderQ::usage="RelationPartialOrderQ[SR] returns True iff the \
relation SR is a partial order."

RelationLinearOrderQ::usage="RelationLinearOrderQ[SR] returns True iff the \
relation SR is a linear order."



KSAxiom1::usage="KSAxiom1[SF] returns True iff the structure SF contains both \
the empty set and the whole set S."

KSAxiom2S::usage="KSAxiom2S[SF]returns True iff the structure SF is stable \
under union."
```

KSAxiom3S::usage="KSAxiom3S[SF]returns True iff the structure SF is stable \
under intersection."

KnowledgeStructureQ::usage="KnowledgeStructureQ[SF]returns True iff the \
structure SF satisfies formal requirements of a knowledge structure."

KnowledgeSpaceQ::usage="KnowledgeSpaceQ[SF]returns True iff the structure SF \
satisfies formal requirements of a knowledge space."

KnowledgeSpaceQuasiOrdinalQ::usage="KnowledgeSpaceQuasiOrdinalQ[SF]returns \
True iff the structure SF is a quasi-ordinal knowledge space."

KSAxiom2C::usage="KSAxiom2C[SF]returns True iff the structure SF is closed \
under union."

KSAxiom3C::usage="KSAxiom3C[SF]returns True iff the structure SF is closed \
under intersection."

FXFilterOrderDown::usage="FXFilterOrderDown[SF,f]returns all members of the \
family forming the structure SF, which are subsets of the family member f."

FXFilterOrderUp::usage="FXFilterOrderUp[SF,f]returns all members of the \
family forming the structure SF, which are supersets of the family member f."

GenerateFamilyByTakingUnions::usage="GenerateFamilyByTakingUnions[F]returns \
the family generated by taking all possible unions on the elements from the \
family F."

AtomsAt::usage="AtomsAt[SF,x]returns all atoms at the element x\[Element]S \
given the structure SF."

Atoms::usage="Atoms[SF]returns subfamilies from the family forming the \
structure SF that are atoms by at least one element from the set forming the \
structure."

BasisFromSpace::usage="BasisFromSpace[SF]returns the basis of the space SF."

BasisToSpace::usage="BasisToSpace[F]returns the space constructed from the \
given basis F."

SSLofClClausesGet::usage="SSLofClClausesGet[LofCl,q]returns the clauses \
assigned to the item q by the mapping LofCl of items to clauses."

SSAttributionQ::usage="SSAttributionQ[LofCl]returns True iff a non-empty \
collection of clauses is assigned to each item within the mapping LofCL of \
items to clauses."

SSAxiom1Q::usage="SSAxiom1Q[LofCl]returns True iff the mapping LofCl of \
items to clauses is an attribution."

SSAxiom2Q::usage="SSAxiom2Q[LofCl]returns True iff each item in the mapping \
LofCL is member of all clauses assigned to it.\n\n
SSAxiom2Q[Q,LofCl]returns True iff each item from Q is member of all clauses \
assigned to it by the mapping LofCL of items to clauses."

SSAxiom2atItemQ::usage="SSAxiom2atItemQ[q,Clauses]returns True iff the item \

```
q is an element of all clauses that are assigned to it in the collection \
Clauses of clauses."

SSAxiom3Q::usage="SSAxiom3Q[LofCl] returns True iff for each clause C of any \
item q given by the mapping LofCl, there exist a clause C1 assigned to each \
item q1 of the clause C, that is a subset of the clause C."

SSAxiom4atClausesQ::usage="SSAxiom4atClausesQ[clauses] returns True iff any \
two clauses for the same item are incomparable."

SSAxiom4Q::usage="SSAxiom4Q[LofCl] returns True iff any two clauses of any \
item, within the mapping LofCl, are incomparable."

SurmiseFunctionQ::usage="SurmiseFunctionQ[f] returns True iff the function f \
represents a surmise function."

SurmiseSystemQ::usage="SurmiseSystemQ[QS] returns True iff QS represents a \
surmise system."

MakeSurmiseFunction::usage="MakeSurmiseFunction[Q,LofCl] returns the data \
structure representing the surmise function given by the set Q of items and \
the mapping LofCl of clauses."

MakeSurmiseSystem::usage="MakeSurmiseSystem[Q,LofCl] returns the data \
structure representing a surmise system given by the set Q of items and the \
mapping LofCl of clauses, under assumption that these objects fulfill the \
necessary formal requirements."

SSSurmiseSet::usage="SSSurmiseSet[SS] returns the set forming the surmise \
system SS."

SSSurmiseFunction::usage="SSSurmiseFunction[SS] returns the surmise function \
forming the surmise system SS."

SSSFListOfClauses::usage="SSSFListOfClauses[SF] returns the list of clauses \
of the surmise function SF."

SSListOfClauses::usage="SSListOfClauses[SS] returns the list of clauses of \
the surmise system SS."

SSSFClausesOfItem::usage="SSSFClausesOfItem[SF,q] returns the clauses \
assigned to the item q by the surmise function SF."

SSClausesOfItem::usage="SSClausesOfItem[SS,q] returns the clauses assigned to \
the item q by the surmise function forming the surmise system SS."

SurmiseSystemToBasis::usage="SurmiseSystemToBasis[SS] returns the basis \
calculated from the surmise system SS."

SurmiseSystemToKnowledgeSpace::usage="SurmiseSystemToKnowledgeSpace[SS]\
returns the basis calculated from the surmise system SS."

SurmiseFunctionFromKnowledgeSpace::usage="SurmiseFunctionFromKnowledgeSpace[\
QK] returns the surmise function calculated from the given knowledge space \
QK."

SurmiseSystemFromKnowledgeSpace::usage="SurmiseSystemFromKnowledgeSpace[QK]\
returns the surmise system representation of the given knowledge space QK."

SurmiseSystemFromBasis::usage="SurmiseSystemFromBasis[B] returns the surmise \
```

```
system calculated from the basis B."

SurmiseFunctionFromBasis::usage="SurmiseFunctionFromBasis[B] returns the \
surmise function calculated from the basis B."



PQxQ::usage="PQxQ[Q] returns the cartesian product between the set Q and its \
powerset from which the empty set was substracted."

PQxPQ::usage="PQxPQ[Q] returns the cartesian product on the powerset of the \
set Q from which the empty set was substracted."

RelationRan::usage="RelationRan[SR,a] returns all elements from the set \
forming the relation SR that are in the relation with the element a."

EntailRelationFromKnowledgeSpace::usage="EntailRelationFromKnowledgeSpace[SF]\
returns the entail relation given the knowledge space SF."

EntailRelationToKnowledgeSpace::usage="EntailRelationToKnowledgeSpace[ESR]\
returns the knowledge space given the entail relation ESR."

(* EntailmentFromKnowledgeSpace[
     SF] returns the entailment from the given knowledge space SF *)

(* EntailmentToKnowledgeSpace[
     ESR] returns the knowledge space from the given entail relation ESR *)

(* ShowsInterestingAssertions[Ent] returns only non-
   redundants assertions of the given entailment Ent *)

EntailmentToEntailRelation::usage="EntailmentToEntailRelation[AAPx] returns \
the entail relation derived from the entailment AAPx."

EntailmentFromEntailRelation::usage="EntailmentFromEntailRelation[AAPB] \
returns the entailment derived from the entail relation AAPB."



Begin["`Private`"]



(* MakeItem[Code,Desc,QText,CorrAns,
     LofDist] returns data object representing an item consisting of unique \
identifier Code, description Desc, text of the question QText,
  correct Answer CorrAns, and the list of distractors LofDist *)
MakeItem[code_,description_,text_,CorrectAnswer_,LofDistractors_List]:=
  Module[{}, {code,description,text,CorrectAnswer,LofDistractors}]

(* ItemCode[Item] returns the code of the given Item *)
ItemCode[item_]:=Module[{},item[[1]]]

(* ItemDescription[Item] returns the description of the given Item *)
ItemDescription[item_]:=Module[{},item[[2]]]

(* ItemText[Item] returns the question-text of the given Item *)
ItemText[item_]:=Module[{},item[[3]]]

ItemCorrectAnswer[item_]:=Module[{},item[[4]]]

ItemAnswers[item_]:=Module[{},item[[5]]]
```

```
(* ItemGet[Items,code] returns the first item its code matches Code *)
ItemGet[Items_,code_]:=Module[{found},
  found:=Select[Items,Function[item,ItemCode[item]===code]];
  If[found=={},{},found[[1]]]
  ]

(* ReadItems[FN,
    "KSMP_ItemsFile"] reads the information about the collection of items \
stored within the file FN according to the fileformat KSMP_ItemsFile *)
ReadItems[FN_String,"KSMP_ItemsFile"]:=
 Module[{concat={},LinesPerItem=5,NofItems,itemsfile,i},
  itemsfile:=DataFileCommentDrop[DataFileRead[FN]];
  NofItems=Length[itemsfile]/LinesPerItem;
  For[i=0,i<LinesPerItem*NofItems,
   AppendTo[concat,
    itemsfile[[Table[j,{j,i+1-LinesPerItem,i}]]]
    ]
   ,i+=LinesPerItem];
  (* Changes the 1st string of each item to INTEGER or SYMBOL *)
  Return[
   Map[(Join[{ToExpression[#[[1]]]},{#[[2]],#[[3]]},{ToExpression[#[[4]]],
         ToExpression[#[[5]]]}])&,concat]]
  ]

(* ReadsItems[
    FN] reads the information about the collection of items stored within \
the file FN according to the default fileformat KSMP_ItemsFile *)
ReadItems[FN_String]:=ReadItems[FN,"KSMP_ItemsFile"]

(* WriteItems[FN,
    Items] stores the information about the Items into the file FN *)
WriteItems[FN_String,Items_List]:=DataFileWrite[FN,Items,"KSMP_ItemsFile"]

(* MakeSkill[Code,Name,
    Desc] returns data object representing an skill consisting of the \
unique identifier Code, thename Name and the description Desc *)
MakeSkill[code_,name_,description_]:=Module[{}, {code,name,description}]

(* SkillCode[Skill] returns the code of the given Skill *)
SkillCode[skill_]:=Module[{},skill[[1]]]

(* SkillName[Skill] returns the name of the given Skill *)
SkillName[skill_]:=Module[{},skill[[2]]]

(* SkillDescription[Skill] returns the description of the given Skill *)
SkillDescription[skill_]:=Module[{},skill[[3]]]

(* SkillGet[Skills,Code] returns the first skill its code matches Code *)
SkillGet[Skills_,code_]:=Module[{found},
  found:=Select[Skills,Function[skill,SkillCode[skill]===code]];
  If[found=={},{},found[[1]]]
  ]

(* ReadSkills[FN,
    "KSMP_SkillsFile"] reads the information about the collection of skills \
stored within the file FN according to the fileformat KSMP_SkillsFile *)
ReadSkills[FN_String,"KSMP_SkillsFile"]:=
 Module[{concat={},LinesPerSkill=3,NofSkills,skillsfile,i},
  skillsfile:=DataFileCommentDrop[DataFileRead[FN]];
```

```
   NofSkills=Length[skillsfile]/LinesPerSkill;
   For[i=0,i<LinesPerSkill*NofSkills,
    AppendTo[concat,
     skillsfile[[Table[j,{j,i+1-LinesPerSkill,i}]]]
     ]
    ,i+=LinesPerSkill];
   (* Changes the 1st string of each skill to INTEGER or SYMBOL *)
   Return[Map[(Join[{ToExpression[#[[1]]]},Rest[#]])&,concat]]
   ]


(* ReadsSkills[
     FN] reads the information about the collection of skills stored within \
the file FN according to the default fileformat KSMP_SkillsFile *)
ReadSkills[FN_String]:=ReadSkills[FN,"KSMP_SkillsFile"]

(* WriteSkills[FN,
     Skills] stores the information about the Skills into the file FN *)
WriteSkills[FN_String,Skills_List]:=DataFileWrite[FN,Skills,"KSMP_SkillsFile"]



(* MakeExpert[MK,CE] returns the data-
   structure representing an expert consisting of domain's metaknowledge MK \
and uncertainty measure CE of making a careless mistake while querying *)
MakeExpert[MetaKnowledge_,CarelessErrors_]:={MetaKnowledge,CarelessErrors}

(* ExpertMetaKnowledge[E] returns expert's E domain metaknowledge  *)
ExpertMetaKnowledge[Expert_]:=Expert[[1]]

(* ExpertCarelessErrors[
     E] returns the uncertainty measure that the expert E makes a careless \
error while querying *)
ExpertCarelessErrors[Expert_]:=Expert[[2]]

(* MakeMetaKnowledge[SF,
     Structure] returns a data structure representing experts knowledge in \
the form of structure for the given knowledge structure SF *)
MakeMetaKnowledge[Structure[S_,F_],Structure]:=Module[{SF=Structure[S,F]},
   Return[SF]]

(* MakeMetaKnowledge[PAss,NAss,UAss,
     Assertions] returns a data structure representing expert's \
metaknowledge in the form of assertions,
  for separately given lists of positive assertions PAss,
  negative assertions Nass and unknown assertions UAss *)
MakeMetaKnowledge[PosAss_,NegAss_,RestAss_,Assertions]:=Module[{},
   Return[Assertions[PosAss,NegAss,RestAss]]
   ]


(* MakeMetaKnowledge[Ent,
     Assertions] returns a data structure representing expert's \
metaknowledge in the form of assertions for the given Entailment Ent.*)
MakeMetaKnowledge[Relation[Q_,PosAss_],Assertions]:=
  Module[{Ent=Relation[Q,PosAss],NegAss},
   NegAss:=SetComplement[PQxQ[Q],PosAss];
   Return[MakeMetaKnowledge[PosAss,NegAss,{},Assertions]]

   ]
```

```
(* MetaKnowledgeAssertionsPositive[Assertions] *)
MetaKnowledgeAssertionsPositive[Assertions[PA_,NA_,UN_]]:=PA

(* MetaKnowledgeAssertionsNegative[Assertions] *)
MetaKnowledgeAssertionsNegative[Assertions[PA_,NA_,UN_]]:=NA

(* MetaKnowledgeAssertionsRest[Assertions] *)
MetaKnowledgeAssertionsRest[Assertions[PA_,NA_,UN_]]:=UN


(* MakeMetaKnowledge[DIAG,
      Diagnostic] returns a data structure representing experts knowledge \
in the form of competence-
   performance diagnostic for the given diagnostic DIAG *)
MakeMetaKnowledge[Diagnostic[E_,K_,A_,P_,k_,p_],Diagnostic]:=Module[
   {DIAG=Diagnostic[E,K,A,P,k,p]},
   Return[DIAG]]
MakeMetaKnowledge[Diagnostic[K_,A_,P_,k_,p_],Diagnostic]:=Module[
   {DIAG=Diagnostic[K,A,P,k,p]},
   Return[DIAG]]

(* MakeSimpleExpert[
     SF] returns an ideal expert possesing domain's metaknowledge given by \
the knowledge space SF *)
MakeSimpleExpert[Structure[S_,F_]]:=Module[{SF=Structure[S,F]},
   MakeExpert[MakeMetaKnowledge[EntailmentFromKnowledgeSpace[SF],Assertions],
    0]]

(* MakeSimpleExpert[
     DIAG] returns an ideal expert possesing domain's metaknowledge given by \
the CPA diagnostic structure DIAG *)
MakeSimpleExpert[DIAG_Diagnostic]:=Module[{},
   MakeExpert[MakeMetaKnowledge[DIAG,Diagnostic],0]
   ]

(* ExpertMakeAnswer[Ass,LV,
     CF] returns a data structure for expert's answer on assertion \
Ass  consisting of the logical value LC and the certainty factor CF *)
ExpertMakeAnswer[Assertion_,LogicalValue_,CertaintyFactor_]:={Assertion,
   LogicalValue,CertaintyFactor}

(* ExpertAnswerLogicalValue[
     Ans] returns the logical value of the expert's answer Ans*)
ExpertAnswerAssertion[Answer_]:=Answer[[1]]

(* ExpertAnswerLogicalValue[
     Ans] returns the logical value of the expert's answer Ans*)
ExpertAnswerLogicalValue[Answer_]:=Answer[[2]]

(* ExpertAnswerCertaintyFactor[
     Ans] returns the certainty factor of the expert's answer Ans *)
ExpertAnswerCertaintyFactor[Answer_]:=Answer[[3]]



MakeStudent[QKnowledge_,CarelessErrors_,LuckyGuesses_]:=
  {QKnowledge,CarelessErrors,LuckyGuesses}

StudentKnowledge[Student_]:=Student[[1]]
```

```
StudentCarelessErrors[Student_]:=Student[[2]]

StudentLuckyGuesses[Student_]:=Student[[3]]

MakeKnowledge[Questions_,Items]:=Questions

MakeSimpleStudent[Questions_]:=MakeStudent[MakeKnowledge[Questions,Items],0,0]

StudentMakeAnswer[question_,LogicalValue_,CertaintyFactor_]:=
  {question,LogicalValue,CertaintyFactor}

StudentAnswerItem[Answer_]:=Answer[[1]]

StudentAnswerLogicalValue[Answer_]:=Answer[[2]]

StudentAnswerCertaintyFactor[Answer_]:=Answer[[3]]



(* RelationReflexiveQ[SR] returns True iff the relation SR is reflexive. *)
RelationReflexiveQ[Relation[S_,R_]]:=ForAllQ[S,(SubsetQ[{{#,#}},R])&]

(* RelationAReflexiveQ[] returns True iff the relation SR is areflexive. *)
RelationAReflexiveQ[SR_Relation]:=¬RelationReflexiveQ[SR]

(* RelationSymmetricQ[SR] returns True iff the relation SR is symmetric *)
RelationSymmetricQ[Relation[S0_,R0_]]:=Module[{S=S0,R=R0},
   ForAllPairsQ[
     S,(SubsetQ[{{#[[1]],#[[2]]}},R]\[Implies]
          SubsetQ[{{#[[2]],#[[1]]}},R])&]
   ]

(* RelationAntiSymmetricQ[
     SR] returns True iff the relation SR is antisymmetric *)
RelationAntiSymmetricQ[Relation[S0_,R0_]]:=Module[{S=S0,R=R0},
   ForAllPairsQ[
     S,((SubsetQ[{{#[[1]],#[[2]]}},R]\[And]
            SubsetQ[{{#[[2]],#[[1]]}},R])\[Implies]#[[1]]===#[[2]])&]
   ]

(* RelationTransitiveQ[SR] returns True if the relation SR is transitive *)
RelationTransitiveQ[Relation[S0_,R0_]]:=Module[{S=S0,R=R0},
   ForAllTriplesQ[S,S,S,
     (SubsetQ[{{#[[1]],#[[2]]}},R]\[And]
          SubsetQ[{{#[[2]],#[[3]]}},R]\[Implies]
         SubsetQ[{{#[[1]],#[[3]]}},R])&
     ]
   ]

(* RelationConnectedQ[
     SR] returns True iff the relation SR is a connected relation *)
RelationConnectedQ[Relation[S0_,R0_]]:=Module[{S=S0,R=R0},
   ForAllPairsQ[
     S,(SubsetQ[{{#[[1]],#[[2]]}},R]\[Or]SubsetQ[{{#[[2]],#[[1]]}},R])&]
   ]

(* RelationEquivalenceQ[
     SR] returns True is the relation SR is an equivalence relation *)
RelationEquivalenceQ[SR_Relation]:=Module[{},
```

```
   RelationReflexiveQ[SR]\[And]RelationSymmetricQ[SR]\[And]
    RelationTransitiveQ[SR]
   ]

(* RelationQuasiOrderQ[SR] returns True if the relation SR is a quasi order *)


RelationQuasiOrderQ[SR_Relation]:=Module[{},
   RelationReflexiveQ[SR]\[And]RelationTransitiveQ[SR]
   ]

(* RelationPartialOrderQ[
    SR] returns True if the relation SR is a partial order *)
RelationPartialOrderQ[SR_Relation]:=Module[{},
   RelationQuasiOrderQ[SR]\[And]RelationAntiSymmetricQ[SR]
   ]

(* RelationLinearOrderQ[
    SR] returns True if the relation SR is a linear order *)
RelationLinearOrderQ[SR_Relation]:=Module[{},
   RelationPartialOrderQ[SR]\[And]RelationConnectedQ[SR]
   ]



\!\(\*
 RowBox[{\( (*\
    KSAxiom1[SF]\ returns\ True\ iff\ the\ structure\ SF\ contains\ both\ \
the\ empty\ set\ and\ the\ whole\ set\ S\ *) \), "\n",
   InterpretationBox[GridBox[{
       {GridBox[{
           {\(KSAxiom1[Structure[S_List, F_List]] :=
              SubsetQ[{{}, S}, F]\)}
           },
          GridBaseline->{Baseline, {1, 1}},
          ColumnWidths->0.999,
          ColumnAlignments->{Left}]}
       },
      GridBaseline->{Baseline, {1, 1}},
      ColumnAlignments->{Left}],
     Definition[ KSAxiom1],
     Editable->False]}]\)



\!\(\*
 RowBox[{\( (*\
    KSAxiom2S[
       SF]\ returns\ True\ iff\ the\ structure\ SF\ is\ stable\ under\ \
union*) \), "\n",
   InterpretationBox[GridBox[{
       {GridBox[{
           {\(KSAxiom2S[Structure[S_List, F_List]] :=
              ForAllPairsQ[F,
               Function[pair,
                Block[{f =
                    pair\[LeftDoubleBracket]1\[RightDoubleBracket],
                   g = pair\[LeftDoubleBracket]2\[RightDoubleBracket]\
}, SubsetQ[{f \[Union] g}, F]]]]\)}
           },
```

```
            GridBaseline->{Baseline, {1, 1}},
            ColumnWidths->0.999,
            ColumnAlignments->{Left}]}
          },
        GridBaseline->{Baseline, {1, 1}},
        ColumnAlignments->{Left}],
      Definition[ KSAxiom2S],
      Editable->False]}]\)



\!\(\*
 RowBox[{\( (*\
     KSAxiom3S[
        SF]\ returns\ True\ iff\ the\ structure\ SF\ is\ stable\ under\ \
intersection\ *) \), "\n",
   InterpretationBox[GridBox[{
        {GridBox[{
            {\(KSAxiom3S[Structure[S_List, F_List]] :=
               ForAllPairsQ[F,
                Function[pair,
                 Block[{f =
                     pair\[LeftDoubleBracket]1\[RightDoubleBracket],
                    g = pair\[LeftDoubleBracket]2\[RightDoubleBracket]\
}, SubsetQ[{f \[Intersection] g}, F]]]]\)}
            },
            GridBaseline->{Baseline, {1, 1}},
            ColumnWidths->0.999,
            ColumnAlignments->{Left}]}
        },
       GridBaseline->{Baseline, {1, 1}},
       ColumnAlignments->{Left}],
      Definition[ KSAxiom3S],
      Editable->False]}]\)



\!\(\*
 RowBox[{\( (*\
     KnowledgeStructureQ[
        SF]\ tests\ whether\ the\ structure\ SF\ satisfies\ formal\ \
requirements\ of\ a\ knowledge\ structure\ *) \), "\n",
   InterpretationBox[GridBox[{
        {GridBox[{
            {\(KnowledgeStructureQ[
                Structure[S_List?SetNonEmptyQ, F_List]] :=
              KSAxiom1[Structure[S, F]]\)}
            },
            GridBaseline->{Baseline, {1, 1}},
            ColumnWidths->0.999,
            ColumnAlignments->{Left}]}
        },
       GridBaseline->{Baseline, {1, 1}},
       ColumnAlignments->{Left}],
      Definition[ KnowledgeStructureQ],
      Editable->False]}]\)



\!\(\*
```

```
 RowBox[{\( (*\
    KnowledgeSpaceQ[
      SF]\ tests\ whether\ the\ structure\ SF\ satisfies\ formal\ \
requirements\ of\ a\ knowledge\ space\ *) \), "\n",
   InterpretationBox[GridBox[{
      {GridBox[{
         {\(KnowledgeSpaceQ[Structure[S_List?SetNonEmptyQ,F_List]] :=
           KSAxiom1[Structure[S, F]] &&
            KSAxiom2C[Structure[S, F]]\)}
         },
        GridBaseline->{Baseline, {1, 1}},
        ColumnWidths->0.999,
        ColumnAlignments->{Left}]}
      },
     GridBaseline->{Baseline, {1, 1}},
     ColumnAlignments->{Left}],
    Definition[ KnowledgeSpaceQ],
    Editable->False]}]\)



(* KnowledgeSpaceQuasiOrdinalQ[
    SF] tests whether the structure SF satisfies formal requirements of a \
knowledge space *)
KnowledgeSpaceQuasiOrdinalQ[Structure[S_List?SetNonEmptyQ,F_List]]:=
 KSAxiom1[Structure[S,F]]&&KSAxiom2C[Structure[S,F]]&&
  KSAxiom3C[Structure[S,F]]

\!\(\*
 RowBox[{\( (*\
    KSAxiom2C[
      SF]\ returns\ True\ iff\ the\ structure\ SF\ is\ closed\ \ under\ \
union*) \), GridBox[{
     {\(KSAxiom2C[Structure[S_, F_]] :=
       Module[{flag = True, N1, N2, subfamiliescollection, i, j},
        N1 = Length[F];
        For[i = 0, i < N1,
         Block[{}, subfamiliescollection = KSubsets[F, i];
          N2 = Length[subfamiliescollection];
          For[j = 0, j < N2,
           Block[{flagtmp1, flagtmp2}, flagtmp1 = flag;
            flagtmp2 =
             SubsetQ[{Union @@
                 subfamiliescollection\[LeftDoubleBracket]
                  j\[RightDoubleBracket]}, F];
            flag = flagtmp1 && flagtmp2;
            If[\(\(\[InvisibleSpace]\)\(!\)\(flag\)\),
             Break[]]; ], \(j++\)];
          If[\(\(\[InvisibleSpace]\)\(!\)\(flag\)\),
           Break[]]; ], \(i++\)]; flag]\)}
     },
    GridBaseline->{Baseline, {1, 1}},
    ColumnWidths->0.999,
    ColumnAlignments->{Left}]}]\)



\!\(\*
 RowBox[{\( (*\
    KSAxiom3C[
```

```
        SF]\ returns\ True\ iff\ the\ structure\ SF\ is\ closed\ \ under\ \
intersection\ *) \), "\n",
   InterpretationBox[GridBox[{
       {GridBox[{
           {\(KSAxiom3C[Structure[S_, F_]] :=
             Module[{flag = True, N1, N2, subfamiliescollection, i,
               j}, N1 = Length[F];
              For[i = 0, i < N1,
               Block[{}, subfamiliescollection = KSubsets[F, i];
                N2 = Length[subfamiliescollection];
                For[j = 0, j < N2,
                 Block[{flagtmp1, flagtmp2}, flagtmp1 = flag;
                  flagtmp2 =
                   SubsetQ[{Intersection @@
                       subfamiliescollection\
\[LeftDoubleBracket]j\[RightDoubleBracket]}, F]; flag = flagtmp1 && flagtmp2;
                  If[\(\(\[InvisibleSpace]\)\(!\)\(flag\)\),
                   Break[]]; ], \(j++\)];
                If[\(\(\[InvisibleSpace]\)\(!\)\(flag\)\),
                 Break[]]; ], \(i++\)]; flag]\)}
           },
          GridBaseline->{Baseline, {1, 1}},
          ColumnWidths->0.999,
          ColumnAlignments->{Left}]}
       },
      GridBaseline->{Baseline, {1, 1}},
      ColumnAlignments->{Left}],
     Definition[ KSAxiom3C],
     Editable->False]]\)



(* FXFilterOrderDown[SF,
     f] returns all the subsets F1 from the structure SF that are subsets of \
the subset f *)
FXFilterOrderDown[Structure[S_,F_],f_]:=Module[{SF=Structure[S,F]},
   Select[F,(SubsetQ[#,f])&]
   ]

(* FXFilterOrderUp[SF,
     f] returns all the subsets F1 from the structure SF that are supersets \
of the subset f *)
FXFilterOrderUp[Structure[S_,F_],f_]:=Module[{SF=Structure[S,F]},
   Select[F,(SubsetQ[f,#])&]
   ]



(* GenerateFamilyByTakingUnions[
     F] returns the family generated by taking unions on the elements from a \
family F *)
GenerateFamilyByTakingUnions[F_List?FamilyQ]:=Module[{j,myset:=F},
   Flatten[Table[
      Map[(Apply[Union,#])&,KSubsets[myset,j]],
      {j,1,Length[myset]}
      ],1]
   ]

(* AtomsAt[SF,
     x] returns the list of atoms of the structure SF at the element x  *)
```

```
AtomsAt[Structure[S_,F_],x0_]:=Module[{SF=Structure[S,F],KQ,RQ1,RQ2},
    KQ:=FX[SF,x0];
    RQ1:={};
    Map[
     Function[pair,Block[{x=pair[[1]],y=pair[[2]]},
        If[SubsetQ[x,y],AppendTo[RQ1,pair]]]]
     ,CartesianProduct[KQ,KQ]];
     RQ2:=MakeRelation[KQ,RQ1];
    MinimalsOfRelation[RQ2]
    ] /; SubsetQ[{x0},S]

(* Atoms[SF] returns all the subfamilies of the family F which are atoms by \
at least one element of the set S. *)
Atoms[Structure[S_,F_]]:=Module[{SF=Structure[S,F]},
  Union[Flatten[Map[(AtomsAt[SF,#])&,S],1]]]

(* BasisFromSpace[SF] calculates the basis of the space SF *)
BasisFromSpace[SF_Structure?KnowledgeSpaceQ]:=Atoms[SF]

(* BasisToSpace[] returns the space for a given basis *)
BasisToSpace[F_List?FamilyQ]:=Module[{F1},
  F1:=Union[Union[GenerateFamilyByTakingUnions[F]],{{}}];
  Return[Structure[Take[F1,-1][[1]],F1]]
  ]



(* SSLofClClausesGet[LofCl,
    q] returns the clauses of the item q that are assigned to it by the \
mapping LofCl of items to clauses *)
SSLofClClausesGet[LofCl_,q_]:=Module[{clauses},
  clauses=Select[LofCl,(#[[1]]==q)&];
  If[clauses\[NotEqual]{},clauses[[1,2]],{}]
  ]

(* SSAttributionQ[LofCl] returns true iff a non-
  empty collection of clauses is assigned to each item within the mapping \
LofCL of items to clauses *)
SSAttributionQ[LofCl_]:=Apply[And,
  Map[Function[pair,SetNonEmptyQ[pair[[2]]]],LofCl]
  ]

(* SSAxiom1Q[
    LofCl] returns true iff  the mapping LofCl of items to clauses is an \
attribution *)
SSAxiom1Q[LofCl_]:=SSAttributionQ[LofCl]

(* SSAxiom2Q[
    LofCl] returns true iff each item from the mapping LofCL is member of \
all clauses assigned to it *)
SSAxiom2Q[LofCl_]:=Module[{Q:=Transpose[LofCl][[1]]},
  SSAxiom2Q[Q,LofCl]]

(* SSAxiom2Q[Q,
    LofCl] returns true iff each item  from Q is member of all clauses \
assigned to it by the mapping LofCL of items to clauses  *)
SSAxiom2Q[Q_,LofCl_]:=ForAllQ[Q,
  Function[item,
    SSAxiom2atItemQ[item,Select[LofCl,(#[[1]]==item)&][[1,2]]]
    ]
```

```
   ]

(* SSAxiom2atItemQ[q,
     clauses] returns true iff the item q is an element of all its assigned \
clauses *)
SSAxiom2atItemQ[q_,clauses_]:=ForAllQ[clauses,(SetElementQ[q,#])&]

(* SSAxiom3Q[
     LofCl] returns true iff for each clause C of any item q given by the \
mapping LofCl,
  there exist some clause C1 assigned to each item q1 of the clause C,
  that is subset of the clause C *)
SSAxiom3Q[LofCl_]:=Module[ {q1},
   Apply[And,
    Map[Function[clause0,
       Block[{clause:=clause0[[2]],C},
        Apply[And,Map[Function[C,
             Apply[And,
              Map[Function[q1,Block[{q1clauses},
                   q1clauses:=SSLofClClausesGet[LofCl,q1];
                   Apply[Or,Map[(SubsetQ[#,C])&,q1clauses]]]],C]
              ]],clause ]]
        ]],LofCl]]]

(* SSAxiom4atClausesQ[
     clauses] returns true iff any two clauses for the same item are \
incomparable *)
SSAxiom4atClausesQ[clauses0_]:=Module[{clauses:=clauses0[[2]]},
   ForAllPairsQ[clauses,
    Function[pair,Block[{C1:=pair[[1]],C2:=pair[[2]]},
       (SubsetQ[C1,C2]\[Implies]EqualSetsQ[C1,C2])
       ]]]]

(* SSAxiom4Q[LofCl] returns true iff any two clauses of any item,
  within the mapping LofCl, are incomparable *)
SSAxiom4Q[LofCl_]:=ForAllQ[LofCl,(SSAxiom4atClausesQ[#])&]

(* SurmiseFunctionQ[
     f] returns true iff the function f represents a surmise function *)
SurmiseFunctionQ[Mapping[Q_,Null,LofCl_]]:=Module[{},
    If[SetNonEmptyQ[Q]\[And] (EqualSetsQ[Q,Transpose[LofCl][[1]]])\[And]
      SSAxiom1Q[LofCl]\[And]SSAxiom2Q[LofCl]\[And]SSAxiom3Q[LofCl]\[And]
      SSAxiom4Q[LofCl],
     True,False]]

(* SurmiseSystemQ[
     QS] returns true iff the Mathematica object QS represents a surmise \
system *)
SurmiseSystemQ[{Q0_,Mapping[Q_,Null,LofCl_]}]:=
  Module[{\[Sigma]=Mapping[Q,Null,LofCl]},
   Apply[And,{Q0==Q,SurmiseFunctionQ[\[Sigma]]}]]

(* MakeSurmiseFunction[Q,
     LofCl] forms the Mathematica object representing the surmise function \
based on the set Q and the mapping LofCl of clauses,
  if these objects fulfill necessary requirements *)
MakeSurmiseFunction[Q_,LofCl_]:=Module[{},
   If[SurmiseFunctionQ[Mapping[Q,Null,LofCl]],Mapping[Q,Null,LofCl]]
   ]
```

```
(* MakeSurmiseSystem[Q,
     LofCl] forms the Mathematica object representing the surmise system \
based on the set Q and the mapping LofCL of clauses,
  if these objects fulfill necessary requirements *)
MakeSurmiseSystem[Q_,LofCl_]:=Module[{},
   If[SurmiseFunctionQ[Mapping[Q,Null,LofCl]],{Q,Mapping[Q,Null,LofCl]}]
   ]

(* SSSurmiseSet[SS] returns the set forming the surmise system SS*)
SSSurmiseSet[SS_]:=SS[[1]]

(* SSSurmiseFunction[
     SS] returns the surmise function forming the surmise system SS *)
SSSurmiseFunction[SS_]:=SS[[2]]


(* SSSFListOfClauses[
     SF] returns the list of clauses of the surmise function SF *)
SSSFListOfClauses[SF_Mapping]:=SF[[3]]

(* SSListOfClauses[SS] returns the list of clauses of the surmise system SS *)


SSListOfClauses[SS_]:=SSSFListOfClauses[SSSurmiseFunction[SS]]

(* SSSFClausesOfItem[SF,
     q] returns the clauses assigned to the item q by the surmise function \
SF *)
SSSFClauseOfItem[SF_,q_]:=
  Select[SSSFListOfClauses[SF],(#[[1]]==q)&][[1,2]]

(* SSClausesOfItem[SS,
     q] returns the clauses assigned to the item q by the surmise function \
forming the surmise system SS *)
SSClausesOfItem[SS_,q_]:=SSSFClauseTake[SSSurmiseFunction[SS],q]

(* SurmiseSystemToBasis[
     SS] returns the basis representation of the surmise system SS *)
SurmiseSystemToBasis[SS_]:= Module[{S:=SSSurmiseSet[SS]},
   Apply[Union,Map[(SSClausesOfItem[SS,#])&,S]]
   ]

(* SurmiseSystemToKnowledgeSpace[
     SS] returns the basis representation of the surmise system SS *)
SurmiseSystemToKnowledgeSpace[SS_]:=BasisToSpace[SurmiseSystemToBasis[SS]]

(* SurmiseFunctionFromKnowledgeSpace[
     QK] returns the surmise function representing the given knowledge space \
QK *)
SurmiseFunctionFromKnowledgeSpace[Structure[Q_,K_]]:=
  Module[{QK:=Structure[Q,K],LofCl},
   LofCl:=Map[({#,AtomsAt[QK,#]})&,Q];
   Return[Mapping[Q,Null,LofCl]]
   ]

(* SurmiseSystemFromKnowledgeSpace[
     QK] returns the surmise system representation of the given knowledge \
space QK *)
SurmiseSystemFromKnowledgeSpace[Structure[Q_,K_]]:=
  Module[{QK:=Structure[Q,K],SSSF},
```

```
   SSSF:=SurmiseFunctionFromKnowledgeSpace[QK];
   Return[{Q,SSSF}]
   ]

(* SurmiseSystemFromBasis[
     B] returns the surmise system equivalent to the knowledge space \
represented by the basis B *)
SurmiseSystemFromBasis[B_]:=Module[{},
   SurmiseSystemFromKnowledgeSpace[BasisToSpace[B]]
   ]


(* SurmiseFunctionFromBasis[
     B] returns the surmise function of the knowledge space represented by \
the basis B *)
SurmiseFunctionFromBasis[B_]:=SSSurmiseFunction[SurmiseSystemFromBasis[B]]



(* PQxQ[Q] returns the cartesian product betwen the set Q
    and its powerset from which the empty set was substracted. *)
PQxQ[X_]:=Module[{},
     SetCartesianProduct[
       SetDifference[PowerSet[X],{{}}],
       X]
     ] /; SetNonEmptyQ[X]

(* PQxPQ[Q] returns the cartesian product on the powerset of the set Q from \
which the empty set was substracted. *)
PQxQ[X_]:=Module[{},
     SetCartesianProduct[
       SetDifference[PowerSet[X],{{}}],
       SetDifference[PowerSet[X],{{}}]
       ]
     ] /; SetNonEmptyQ[X]

(* RelationRan[SR,
     a] returns the set of all elements for which holds that the element a \
is in the relation SR with them. *)
RelationRan[Relation[S_,R_],a_]:=
  Map[(#[[2]])&,Select[R,(#[[1]]==a)&]] /; SubsetQ[{a},S]

(* EntailRelationFromKnowledgeSpace[
     SF] returns the entail relation from the given knowledge space SF *)
EntailRelationFromKnowledgeSpace[Structure[S_,F_]]:=Module[{},
   Relation[S,Select[PQxPQ[S],
       Function[{AB},ForAllQ[F,
          ((Intersection[AB[[1]],#]==={})\[Implies](Intersection[
                AB[[2]],#]==={}))&
          ]]
       ] ]
   ]


(* EntailRelationToKnowledgeSpace[
     ESR] returns the knowledge space from the given entail relation ESR *)
EntailRelationToKnowledgeSpace[Relation[S_,R_]]:=Module[{},
   Structure[S,
     Select[PowerSet[S],
       Function[{K},
```

```
        ForAllQ[R,
          Function[AB,(Intersection[K,AB[[1]]]==={})\[Implies]
            (Intersection[K,AB[[2]]]==={})]
          ]]]]]

(* EntailmentFromKnowledgeSpace[
    SF] returns the entailment from the given knowledge space SF *)
EntailmentFromKnowledgeSpace[Structure[S_,F_]]:=Module[{},
  Relation[S,Select[PQxQ[S],
    Function[{AB},ForAllQ[F,
      ((Intersection[AB[[1]],#]==={})\[Implies](¬SetElementQ[
              AB[[2]],#]))&
      ]]
    ] ]
  ]


(* EntailmentToKnowledgeSpace[
    ESR] returns the knowledge space from the given entail relation ESR *)
EntailmentToKnowledgeSpace[Relation[S_,R_]]:=Module[{},
  Structure[S,
   Select[PowerSet[S],
    Function[{K},
     ForAllQ[R,
      Function[AB,(Intersection[K,AB[[1]]]==={})\[Implies]
        (¬SetElementQ[AB[[2]],K])]
      ]]]]]

(* ShowsInterestingAssertions[Ent] returns only non-
  redundants assertions of the given entailment Ent *)
ShowInterestingAssertions[Entailment_]:=Module[{},
  Select[Entailment[[2]],(¬SetElementQ[#[[2]],#[[1]]])&]
  ]

(* EntailmentToEntailRelation[
    AAPx] returns the entail relation derived from the entailment AAPx *)
EntailmentToEntailRelation[Relation[S_,ENT_]]:=Module[{As,A,B0,q,Bs},
  As:=Union[Transpose[ENT][[1]]];
  Bs :=Apply[Join,
    Map[Function[A,Block[{},
        B0=Map[(#[[2]])&,Select[ENT,(Block[{B=#[[1]]},B==A])&]];
        Map[({A,#})&,SetComplement[PowerSet[B0],{{}}]]
        ]],As]
    ];
  Return[Relation[S,Bs]]
  ]

(* EntailmentFromEntailRelation[
    AAPB] returns the entailment derived from the entail relation AAPB *)
EntailmentFromEntailRelation[Relation[Q_,E_]]:=Module[{As,P},
  As:=Union[Transpose[E]][[1]];
  P:=Apply[Union,
    Map[Function[A,Block[{B0},
        (* za dani A formiraj B0 *)
        B0=Apply[Union,Apply[Union,
            Select[E,(Block[{A1=#[[1]]},A1==A])&]]];
        Map[Function[b,{A,b}],B0]
        ]],As]
    ];
  Return[Relation[Q,P]]
```

```
  ]

End[]

EndPackage[]
```

KSMP skills module

```
(* :Title: Knowledge Spaces Mathematica Package - Skills Module. *)

(* :Context: KnowledgeSpaces`Skills` *)

(* :Author: Andrej Zaluski *)

(* :Summary:
     This package contains the Skills Module of the KnowledgeSpaces
      Mathematica package. The Skills Module implements various approaches
      for handling latent skills and competencies within the
      field of knowledge spaces.
 *)

(*  :Copyright: 20001, Andrej Zaluski.
          See attached license.
 *)

(*  :Package Version: 0.9 beta 1 *)

(*  :Mathematica Version: 4.0 *)

(*  :History:
            1.0 reserved for the first public version.
            0.9 beta 1 version. *)

(*  :Keywords:
     Psychology and Education; Mathematical Psychology;
  Knowledge Modeling; Knowledge Space Theory; Skills Modeling;
 *)

(*  :Sources: *)

(*  :Warnings: BETA 1 - TESTING PHASE. *)

(*  :Limitations:
 *)

(*  :Discussion: *)

(*  :Requirements:
                    Packages: KnowledgeSpaces`L2`
  *)



(* TO DO *) (* Add also KSMP_Skills.m? *)
BeginPackage["KnowledgeSpaces`Skills`",{"KnowledgeSpaces`L2`"}]

(* TODO *) (* Save the state of the package *)
Off[General::spell,General::spell1]


CSAxiom1Q::usage="CSAxiom1Q[EK] returns True iff the structure EK fulfills \
the property \[Union]K=E."

PSAxiom1Q::usage="PSAxiom1Q[AP] returns True iff the structure AP fulfills \
the property \[Union]P=A."
```

```
CSAxiom2Q::usage="CSAxiom2Q[EK] returns True iff the structure fulfills the \
property that
  the family K contains both the empty set and the set E."

PSAxiom2Q::usage="PSAxiom2Q[AP] returns True iff the structure AP fulfills \
the property that
  the family P contains both the empty set and the set A."

CSAxiom3Q::usage="CSAxiom3Q[EK] returns True iff the structure EK is stable \
under union."

PSAxiom3Q::usage="PSAxiom3Q[AP] returns True iff the structure AP is stable \
under union."

CompetenceStructureQ::usage="CompetenceStructureQ[EK] returns True iff the \
structure EK is
  a competence structure."

PerformanceStructureQ::usage="PerformanceStructureQ[AP] returns True iff the \
structure AP is
  a performance structure."

CompetenceSpaceQ::usage="CompetenceSpaceQ[EK] returns True iff the structure \
EK is
  a competence space."

PerformanceSpaceQ::usage="PerformanceSpaceQ[AP] returns True iff the \
structure AP is
  a performance space."

CPAMakeInterpretationFunction::usage="CPAMakeInterpretationFunction[A,K,KXs] \
returns the interpretation function given the set A of items, the competence \
structure K, and the mapping KXs interpreting each item by assigning \
competence states."

CPAInterpretationFunctionInitialize::usage=\
"CPAInterpretationFunctionInitialize[K] returns the empty interpretation \
function given the competence space K and the empty set of problems.\n\n
CPAInterpretationFunctionInitialize[A,K] returnes the initialized \
interpretation function consisting of the non-interpreted items from the set \
A of items given the competence space K."

CPAInterpretationFunctionAxiomsQ::usage="CPAInterpretationFunctionAxiomsQ[k] \
returns True iff the function k satisfies the axiom 1 of interpretation \
functions."

CPAInterpretationFunctionQ::usage="CPAInterpretationFunctionQ[f] returns True \
iff the function f is an interpretation function."

CPAInterpretationFunctionItems::usage="CPAInterpretationFunctionItems[k] \
returns the set of items contained in the interpretation function k."

CPAInterpretationFunctionInterpretations::usage=\
"CPAInterpretationFunctionInterpretations[k] returns the list of \
interpretations contained in the interpretation function k."

CPAInterpretationsInterpretationReplace::usage=\
"CPAInterpretationsInterpretationReplace[KXs,NewKx] returns the
  interpretations KXs after replacing the corresponding interpretation with \
```

```
the interpretation NewKx."

CPAInterpretationsInterpretationAdd::usage=\
"CPAInterpretationsInterpretationAdd[KXs,kx] returns the interpretations KXs \
after adding the interpretation kx."

CPAInterpretationFunctionUpgrade::usage="CPAInterpretationFunctionUpgrade[k,I]\
 returns the  interpretation function k upgraded with the (possibly empty)
   list I of interpretations or an individual interpretation I."


CPAInterpretationsDeleteItemInterpretations::usage=\
"CPAInterpretationsDeleteItemInterpretations[KXs,q] returns the list of \
interpratations after deleting all interpretations on the item q from the \
list KXs of interpretations."

CPAInterpretationFunctionConformTo::usage="CPAInterpretationFunctionConformTo[\
k] returns the interpretation function obtained from the given interpretation \
function k after reducing its set of items to a competence-based item set."

CPAInterpretationOfItem::usage="CPAInterpretationOfItem[k,x]returns the \
interpretation of the item x given the interpretation function k."

CPARepresentationOfCompetenceState::usage="CPARepresentationOfCompetenceState[\
k,C] returns the data structure {C,p(C)}, where p(C) is a set of items \
assigned by the interpretation function k to the given competence state C."

CPARepresentations::usage="CPARepresentations[k] returns the list of all \
representations of the competence structure derived from
  the interpretation function k."

CPARepresentationFunctionFromInterpretationFunction::usage=\
"CPARepresentationFunctionFromInterpretationFunction[k] returns the \
representation function given the interpretation function k."

CPAMakeRepresentation::usage="CPAMakeRepresentation[C,Items] returns the data \
structure containing the representation formed by the given competence state \
C and the set Items."

CPARepresentationCompetenceState::usage="CPARepresentationCompetenceState[PC] \
returns the competence state
  of the given representation PC."

CPARepresentationItems::usage="CPARepresentationItems[pC] returns the set of \
items for the given representation pC."

CPAMakeRepresentationFunction::usage="CPAMakeRepresentationFunction[K,A,PCs] \
returns the data structure
  containing the representation function formed from the competence structure \
K, the set A of items and the collection PCs of representations."

CPARepresentationFunctionCompetenceStructure::usage=\
"CPARepresentationFunctionCompetenceStructure[p] returns the competence
  structure forming the given representation function p."

CPARepresentationFunctionRepresentations::usage=\
"CPARepresentationFunctionRepresentations[p] returns the collection
   of representations forming the representation function p."

CPAMakeDiagnostic::usage="CPAMakeDiagnostic[K,A,P,k,p]returns the data \
```

```
structure representing the diagnostic given by the family forming the \
competence structure K, the set A of items, the family forming the \
performance structure P, the interpretation function k and the representation \
function p.\n\n
CPAMakeDiagnostic[E,K,A,P,k,p]returns the data structure representing the \
diagnostic given by the set E of elementary competencies, the family forming \
the competence structure K, the set A of items, the family forming the \
performance structure P, the interpretation
function k and the representation function p."

CPADiagnosticElementaryCompetencies::usage=\
"CPADiagnosticElementaryCompetencies[Diag] returns the set of elementary \
competencies forming the given diagnostic Diag."

CPADiagnosticCompetenceSpace::usage="CPADiagnosticCompetenceSpace[Diag] \
returns the competence space forming the given diagnostic Diag."

CPADiagnosticItems::usage="CPADiagnosticItems[Diag] returns the set of items \
forming the given diagnostic Diag."

CPADiagnosticPerformanceSpace::usage="CPADiagnosticPerformanceSpace[Diag] \
returns the performance space forming the given diagnostic Diag."

CPADiagnosticInterpretationFunction::usage=\
"CPADiagnosticInterpretationFunction[Diag] returns the interpretation \
function forming the given diagnostic Diag."

CPADiagnosticRepresentationFunction::usage=\
"CPADiagnosticRepresentationFunction[Diag] returns the representation \
function forming the given diagnostic Diag."

CPAEqualDiagnosticsQ::usage="CPAEqualDiagnosticsQ[Diag1,Diag2] returns True \
iff the diagnostics
  Diag1 and Diag2 are equal."

CPARepresentationStructureFromInterpretationFunction::usage=\
"CPARepresentationStructureFromInterpretationFunction[k] returns the induced \
performance structure from the given interpretation function k."

CPARepresentationStructureQ::usage="CPARepresentationStructureQ[AP,k] returns \
True iff the given performance structure AP is the representation structure \
under the interpretation function k."

CPADiagnosticFromInterpretationFunction::usage=\
"CPADiagnosticFromInterpretationFunction[k] returns the (6-tupel) diagnostic \
induced by the interpretation function k. It is assumed that the competence \
structure given by k is based on a set of elementary competencies."


Begin["`Private`"]


(* CSAxiom1Q[EK] returns True iff the structure fulfills \[Union]K=
   E property *)
CSAxiom1Q[Structure[E_List?SetNonEmptyQ,K_List?FamilyQ]]:=Module[{},
   EqualSetsQ[Apply[Union,K],E]]

(* PSAxiom1Q[AP] returns True iff the structure fulfills \[Union]P=
   A property *)
PSAxiom1Q[Structure[A_List?SetNonEmptyQ,P_List?FamilyQ]]:=Module[{},
```

```
   EqualSetsQ[Apply[Union,P],A]]

(* CSAxiom2Q[EK] returns True iff the structure fulfills the property that
   the family K contains both {} and E as its subsets *)
CSAxiom2Q[Structure[E_List?SetNonEmptyQ,K_List?FamilyQ]]:=
 Module[{},SubsetQ[{{},E},K]]

(* CSAxiom2Q[EK] returns True iff the structure fulfills the property that
   the family K contains both {} and E as its subsets *)
CSAxiom2Q[Structure[E_List?SetNonEmptyQ,K_List?FamilyQ]]:=
 Module[{},SubsetQ[{{},E},K]]

(* PSAxiom2Q[AP] returns True iff the structure fulfills the property that
   the family P contains both {} and A as its subsets *)
CSAxiom2Q[Structure[A_List?SetNonEmptyQ,P_List?FamilyQ]]:=
 Module[{},SubsetQ[{{},A},P]]

(* CSAxiom3Q[
    EK] returns True iff the structure EK fulfills the property of being \
stable under union *)
CSAxiom3Q[Structure[E_List?SetNonEmptyQ,K_List?FamilyQ]]:=Module[{},
   ForAllPairsQ[K,
    Function[pair,Block[{f=pair[[1]],g=pair[[2]]},
       SubsetQ[{f\[Union]g},K]
       ]]]]

(* PSAxiom3Q[
    EK] returns True iff the structure EK fulfills the property of being \
stable under union *)
PSAxiom3Q[Structure[A_List?SetNonEmptyQ,P_List?FamilyQ]]:=Module[{},
   ForAllPairsQ[P,
    Function[pair,Block[{f=pair[[1]],g=pair[[2]]},
       SubsetQ[{f\[Union]g},P]
       ]]]]

(* CompetenceStructureQ[EK] returns True iff the structure EK is
   a competence structure *)
CompetenceStructureQ[Structure[E_List?SetNonEmptyQ,K_List?FamilyQ]]:=
 Module[{EK=Structure[E,K]}, CSAxiom1Q[EK]]

(* PerformanceStructureQ[AP] returns True iff the structure AP is
   a performance structure *)
PerformanceStructureQ[Structure[E_List?SetNonEmptyQ,K_List?FamilyQ]]:=
 Module[{EK=Structure[E,K]}, PSAxiom1Q[EK]]

(* CompetenceSpaceQ[EK] returns True iff the structure EK is
   a competence space *)
CompetenceSpaceQ[Structure[E_List?SetNonEmptyQ,K_List?FamilyQ]]:=
 Module[{EK=Structure[E,K]}, CSAxiom1Q[EK]&&CSAxiom2Q[EK]&&CSAxiom3Q[EK]
   ]

(* PerformanceSpaceQ[AP] returns True iff the structure AP is
   a competence space *)
PerformanceSpaceQ[Structure[E_List?SetNonEmptyQ,K_List?FamilyQ]]:=
 Module[{EK=Structure[E,K]}, CSAxiom1Q[EK]&&CSAxiom2Q[EK]&&CSAxiom3Q[EK]
   ]

(* CPAMakeInterpretationFunction[A,K,
    KXs] returns the interpretation function give the set A of items,
 the competence structure K,
```

```
  and the mapping KXs interpreting each item by assigning competence states *)


CPAMakeInterpretationFunction[A_,K_,KXs_]:=Module[{f},
   f=MakeFunction[A,PowerSet[K],KXs];
   If[FunctionWellDefinedQ[f],Return[f]]
   ]


(* CPAInterpretationFunctionInitialize[
     K] returns empty interpretation function for given competence space K \
and empty set of problems*)
CPAInterpretationFunctionInitialize[K_]:=Module[{},
   MakeFunction[{},PowerSet[K],{}]
   ]

(* CPAInterpretationFunctionInitialize[A,
       K] returnes initialized interpretation function consisting of non-
   interpreted items from the set A of problems given the competence space \
K. *)
CPAInterpretationFunctionInitialize[A_,K_]:=Module[{},
   MakeFunction[A,PowerSet[K],Map[({#,{}})&,A]]
   ]

(* CPAInterpretationFunctionAxiomsQ[
     k] returns True iff the function k satisfies the axiom 1 of \
interpretation functions,
  i.e. all the items are interpreted and informative. *)
CPAInterpretationFunctionAxiomsQ[Mapping[D_,C_,M_]]:=
  Module[{f=Mapping[D,C,M],KXs,K},
   K=Apply[Union,C];
   KXs=FunctionMappingImage[f];
   Apply[And,
    Map[Function[kx,(¬EqualSetsQ[kx,{}]\[And]¬EqualSetsQ[K,kx])],
      KXs]]
   ]

(* CPAInterpretationFunctionQ[
     f] returns True iff the function f is an interpretation function *)
CPAInterpretationFunctionQ[Mapping[D_,C_,M_]]:=Module[
   {f=Mapping[D,C,M]},
   FunctionQ[f]\[And]FunctionWellDefinedQ[f]\[And]
    CPAInterpretationFunctionAxiomsQ[f]
   ]

(* CPAInterpretationFunctionItems[
     k] returns the set of items contained in the interpretation function k *)


CPAInterpretationFunctionItems[Mapping[A_,PsK_,KXs_]]:=A;

(* CPAInterpretationFunctionInterpretations[
     k] returns the list of interpretations contained in the interpretation \
function k *)
CPAInterpretationFunctionInterpretations[Mapping[A_,PsK_,KXs_]]:=KXs;

(* CPAInterpretationsInterpretationReplace[KXs,NewKx] returns the
   interpretations KXs with the replaced interpretation NewKx
   for the corresponding item  *)
CPAInterpretationsInterpretationReplace[KXs_,NewKx_]:=Module[{item},
```

```
   item=NewKx[[1]];
   If[Select[KXs,(#[[1]]===item)&]\[NotEqual]{},
    ReplacePart[KXs,NewKx,Position[KXs,
       Select[KXs,(#[[1]]===item)&][[1]]]],
    KXs]
   ]

(* CPAInterpretationsInterpretationAdd[KXs,
    kx] returns the interpretations KXs with added interpretation kx  *)
CPAInterpretationsInterpretationAdd[KXs_,kx_]:=Module[{item},
   item=kx[[1]];
   If[Select[KXs,(#[[1]]===item)&]=={},
    Union[KXs,{kx}]
    ,KXs
    ]
   ]

(* CPAInterpretationFunctionUpgrade[k,
    I] returns the  interpretation function k upgraded with the (possibly \
empty)
    list I of interpretations or an individual interpretation I *)


CPAInterpretationFunctionUpgrade[k_,I_]:=Module[{KXs,kx},
   Which[I=={},Return[k]
    ,(Head[I[[1]]]===List),
    Return[CPAInterpretationFunctionUpgradeAux1[k,I]];
    ,(MemberQ[{Symbol,Integer,String},Head[I[[1]]]]==True),
    Return[CPAInterpretationFunctionUpgradeAux2[k,I]];
    ];
   ]

(* HIDDEN *)
CPAInterpretationFunctionUpgradeAux1[k_,KXs_]:=Module[{NofIASSs,i},
   NofIASSs=Length[KXs];
   For[i=1,i<=NofIASSs,i++,
    k=CPAInterpretationFunctionUpgradeAux2[k,KXs[[i]]]];
   Return[k]
   ]


(* HIDDEN *)
CPAInterpretationFunctionUpgradeAux2[Mapping[A_,PsK_,KXs_],kx_]:=
  Module[{k=Mapping[A,PsK,KXs],A1=A,KXs1=KXs,K,a,kx0,kx1},
   a=kx[[1]];
   K=Apply[Union,PsK];
   (* Check whether the interpretation function axioms are
      fulfilled for kx *)
   If[(¬EqualSetsQ[kx,{}]\[And]¬EqualSetsQ[kx,K]),
    A1=Union[A,{a}];
    (* Check whether a partial intepretation kx already exists? *)
    kx0=Select[KXs,(#[[1]]===a)&];
    If[kx0=={},
     (* No, simply add new interpretation*)
     KXs1=CPAInterpretationsInterpretationAdd[KXs,kx];,
     (* Yes, join old partial and new partial interpretation *)
      kx1={kx0[[1,1]],Union[ kx0[[1,2]],kx[[2]]]};
     KXs1=CPAInterpretationsInterpretationReplace[KXs,kx1];
     ];
    ];
```

```
   Return[Mapping[A1,PsK,KXs1]]
   ]

(* CPAInterpretationsDeleteItemInterpretations[KXs,
     q] returns the list of interpratations where all the interpretations on \
the item q were removed from the original list KXs of interpretations *)
CPAInterpretationsDeleteItemInterpretations[KXs0_,q_]:=Module[{KXs},
   KXs=Select[KXs0,(#[[1]]==q)&];
   Return[SetDifference[KXs0,KXs]]
   ]

(* CPAInterpretationFunctionConformTo[
      k] returns the interpretation function obtained from the given \
interpretation function k by reducing its set of items to a competence-
   based item set,
 i.e. by eliminating non-interpreted and non-informative interpretations. *)


CPAInterpretationFunctionConformTo[Mapping[A0_,PK_,KXs0_]]:=
 Module [{K=Apply[Union,PK],KXs,A},
   K=Apply[Union,PK];
   (* select interpretations not conforming to the interpretation
      function definition *)
   KXs=Select[KXs0,((SetEmptyQ[#[[2]]])\[Or](EqualSetsQ[#[[2]],K]))&];
   (* create the list containing only conforming assertions *)
   KXs=SetDifference[KXs0,KXs];
   (* form the competence-based item set *)
   A=Union[Map[(#[[1]])&,KXs]];
   (* return the interpretation function based on the competence
      based item set *)
   Return[Mapping[A,PK,KXs]]
   ]

(* CPAInterpretationOfItem[k,
     x] returns the interpretation of the item x given the interpretation \
function k *)
CPAInterpretationOfItem[Mapping[A_,PK_,KXs_],x_]:=Module[{int},
   int=Select[KXs,(#[[1]]==x)&,1];
   If[int\[NotEqual]{},int[[1,2]],{}]
   ]

(* CPARepresentationOfCompetenceState[k,C] returns the data structure {C,
     p(C)}, where p(C) is the subset of all the items (induced the \
interpretation function k) that represent the given competence state C *)
CPARepresentationOfCompetenceState[Mapping[A_,PK_,KXs_],C_]:=
 Module[{k=Mapping[A,PK,KXs]},
   Return[{C,Select[A,(SetElementQ[C,CPAInterpretationOfItem[k,#]])&]}]
   ]

(* CPARepresentations[k] returns the list of all representations on the
   set of items interpreted in thecompetence structure given through
   the interpretation function k *)
CPARepresentations[Mapping[A_,PK_,KXs_]]:=
 Module[{k=Mapping[A,PK,KXs],K=Apply[Union,PK]},
   Map[(CPARepresentationOfCompetenceState[k,#])&,K]
   ]

(* CPARepresentationFunctionFromInterpretationFunction[
     k] returns the representation function from the given interpretation \
function k  *)
```

```
CPARepresentationFunctionFromInterpretationFunction[Mapping[A_,PK_,KXs_]]:=
  Module[{k=Mapping[A,PK,KXs],K=Apply[Union,PK]},
    Return[Mapping[K,PowerSet[A],CPARepresentations[k]]]
    ]

(* CPAMakeRepresentation[C,
     Items] returns the data structure containing the representation formed \
by the given competence state C and the Items set *)
CPAMakeRepresentation[C0_,Items_]:={C0,Items}

(* CPARepresentationCompetenceState[PC] returns the competence state
   of the given representation PC  *)
CPARepresentationCompetenceState[PC_]:=PC[[1]]

(* CPARepresentationItems[
     pC] returns the items of the given representation pC *)
CPARepresentationItems[r_List]:=r[[2]]

(* CPAMakeRepresentationFunction[K,A,PCs] returns the data structure
   containing the representation function formed from the competence \
structure K, the set A of problems and the collection PCs of representations *)


CPAMakeRepresentationFunction[K_,A_,PCs_]:=Mapping[K,PowerSet[A],PCs]

(* CPARepresentationFunctionCompetenceStructure[p] returns the competence
   structure forming the given  representation function p *)
CPARepresentationFunctionCompetenceStructure[Mapping[K_,PA_,PCs_]]:=K;

(* CPARepresentationFunctionRepresentations[p] returns the collection
     of representations forming the  *)
CPARepresentationFunctionRepresentations[Mapping[K_,PA_,PCs_]]:=PCs;

(* CPAMakeDiagnostic[K,A,P,k,
     p]returns the data structure representing the diagnostic given by the \
family forming the competence structure K, the set A of problems,
  the family forming the performance structure P,
  the interpretation function k and the representation function p *)
CPAMakeDiagnostic[K_,A_,P_,k_,p_]:=Diagnostic[K,A,P,k,p]

(* CPAMakeDiagnostic[E,K,A,P,k,
     p]returns the data structure representing the diagnostic given by the \
set E of elementary competencies,
  the family forming the competence structure K,the set A of problems,
  the family forming performance structure P, the interpretation
    function k and the representation function p *)
CPAMakeDiagnostic[E_,K_,A_,P_,k_,p_]:=Diagnostic[E,K,A,P,k,p]

(* CPADiagnosticElementaryCompetencies[
     Diag] returns the set of elemntary competencies forming the given \
diagnostic Diag *)
CPADiagnosticElementaryCompetencies[Diagnostic[E_,_,_,_,_,_]]:=E;

(* CPADiagnosticCompetenceSpace[
     Diag] returns the competence space forming the given diagnostic Diag *)
CPADiagnosticCompetenceSpace[Diagnostic[K_,_,_,_,_]]:=K;
CPADiagnosticCompetenceSpace[Diagnostic[_,K_,_,_,_,_]]:=K;

(* CPADiagnosticItems[
     Diag] returns the set of items forming the given diagnostic Diag *)
```

```
CPADiagnosticItems[Diagnostic[_,A_,_,_,_]]:=A;
CPADiagnosticItems[Diagnostic[_,_,A_,_,_,_]]:=A;

(* CPADiagnosticPerformanceSpace[
     Diag] returns the performance space forming the given diagnostic Diag *)


CPADiagnosticPerformanceSpace[Diagnostic[_,_,P_,_,_]]:=P;
CPADiagnosticPerformanceSpace[Diagnostic[_,_,_,P_,_,_]]:=P;

(* CPADiagnosticInterpretationFunction[
     Diag] returns the interpretation function forming the given diagnostic \
Diag *)
CPADiagnosticInterpretationFunction[Diagnostic[_,_,_,k_,_]]:=k;
CPADiagnosticInterpretationFunction[Diagnostic[_,_,_,_,k_,_]]:=k;

(* CPADiagnosticRepresentationFunction[
     Diag] returns the representation function forming the given diagnostic \
Diag *)
CPADiagnosticRepresentationFunction[Diagnostic[_,_,_,_,p_]]:=p;
CPADiagnosticRepresentationFunction[Diagnostic[_,_,_,_,_,p_]]:=p;

(* CPAEqualDiagnosticsQ[Diag1,Diag2] returns True iff the diagnostics
   Diag1 and Diag2 are equal *)
CPAEqualDiagnosticsQ[Diagnostic[E1_,K1_,A1_,P1_,k1_,p1_],
   Diagnostic[E2_,K2_,A2_,P2_,k2_,p2_]]:=EqualSetsQ[E1,E2]&&
   EqualSetsQ[K1,K2]&&EqualSetsQ[A1,A2]&&EqualSetsQ[P1,P2]&&
   EqualFunctionsQ[k1,k2]&&EqualFunctionsQ[p1,p2]

(* CPARepresentationStructureFromInterpretationFunction[
     k] returnes the induced performance structure from the given \
interpretation function k *)
CPARepresentationStructureFromInterpretationFunction[Mapping[A0_,PK0_,KXs0_]]:=
  Module[{k=Mapping[A0,PK0,KXs0],K=Apply[Union,PK0],P},
   P=Union[Map[(CPARepresentationOfCompetenceState[k,#][[2]])&,K]];
   Return[Structure[A,P]]
   ]

(* CPADiagnosticFromInterpretationFunction[
     k] returns the (6-
       tupel) diagnostic induced by the interpretation function k.
     It is assumed that is the competence structure given through k is \
represented using a set of elementary competencies*)
CPADiagnosticFromInterpretationFunction[Mapping[A_,PK_,KXs_]]:=
  Module[{E,K=Apply[Union,PK],P,k=Mapping[A,PK,KXs],p},
   E=Apply[Union,K];
   p=CPARepresentationFunctionFromInterpretationFunction[k];
   P=CPARepresentationStructureFromInterpretationFunction[k];
   Return[CPAMakeDiagnostic[E,K,A,P[[2]],k,p]];
   ]

End[]
EndPackage[]
```

KSMP querying module

```
(* :Title: Knowledge Spaces Mathematica Package - Querying Module. *)

(* :Context: KnowledgeSpaces`Querying` *)

(* :Author: Andrej Zaluski *)

(* :Summary:

 *)

(*  :Copyright: 20001, Andrej Zaluski.
        See attached license.
 *)

(*  :Package Version: 0.9 beta 1 *)

(*  :Mathematica Version: 4.0 *)

(*  :History:
          1.0 reserved for the first public version.
          0.9 beta 1 version. *)

(*  :Keywords:
    Psychology and Education; Mathematical Psychology;
 Knowledge Modeling; Knowledge Space Theory; Querying Experts;
 *)

(*  :Sources: *)

(*  :Warnings: *)

(*  :Limitations:
 *)

(*  :Discussion: *)

(*  :Requirements:
                Packages: KnowledgeSpaces`Skills`
 *)



BeginPackage["KnowledgeSpaces`Querying`",{"KnowledgeSpaces`L1`","KnowledgeSpaces`L2`","KnowledgeSpaces`Skills`"}]

(* TODO *) (* Save the state of the package *)
Off[General::spell,General::spell1]




QEKAssertionPremise::usage="QEKAssertionPremise[Assertion]returns the \
premise contained in the assertion Assertion."

QEKAssertionConsequence::usage="QEKAssertionConsequence[Assertion]returns \
the consequence contained in the assertion Assertion."
```

```
QEKMakeAssertions::usage="QEKMakeAssertions[Q,Premise] returns all possible \
assertions given the premise Premise and the set Q of items, assuming that \
only one element in the consequence of the assertion is present."

QEKMakeFullBlock::usage="QEKMakeFullBlock[Q,k] returns the block (list of the \
assertions)
  of the subsets-by-items table defined by the given premise size k and the \
set Q of items."

QEKMakeSbITable::usage="QEKMakeSbITable[Q]returns the subsets-by-items table \
given the set Q of items."

QEKQueryTexts::usage="QEKQueryTexts[PremiseSize,TextPart]returns the partial \
text of a query on an assertion given the text part TextPart (introduction or \
question) and the premise size PremiseSize (1,2,generic,previous_to_last)."

QEKFormulateQuery::usage="QEKFormulateQuery[Q,Items,Assertion]returns the \
query question on the given assertion Assertion formulated for a human \
expert, given the set Q of items described in Items."

QEKAskExpert::usage="QEKAskExpert[Q,Items,Assertion,Human]returns an answer \
from a human experts on assertion Assertion containing some items from the \
set Q described in Items.\n\n
QEKAskExpert[Assert,Exp,Simulation] returns an answer from the simulated \
expert Exp on the assertion Assert. The expert's domain meta-knowledge may be \
represented either using a structure or a diagnostic."

QEKExtractPositiveJudgments::usage="QEKExtractPositiveJudgments[QH] returns \
all positive judgments from the querying history QH."

QEKExtractNegativeJudgments::usage="QEKExtractNegativeJudgments[QH] returns \
all negative judgments from the querying history QH."

QEKQueryExpertStraightforward::usage="QEKQueryExpertStraightforward[Q,Items,\
Human] returns the jugdments obtained while querying a human expert on \
structure of the items Q described in Items.\n\n
QEKQueryExpertStraightforward[Q,Expert,Simulation] returns the jugdments \
obtained while querying the simulated Expert on structure of the items Q."

QEKEntailmentFromQueryHistory::usage="QEKEntailmentFromQueryHistory[Q,\
QueryHistory] returns all positive judgments from the querying history \
QueryHistory capturing an establishment of a structure on the set Q of items."

QEKEstablishKnowledgeSpaceStraightforward::usage=\
"QEKEstablishKnowledgeSpaceStraightforward[Q,Items,Human] returns a structure \
on the items Q described in Items, established by querying a human Expert.\n\n\

QEKEstablishKnowledgeSpaceStraightforward[Q,Expert,Simulation] returns a \
structure on the items Q obtained by querying the simulated expert Expert."

QEKAssertionsAxiom1Drop::usage="QEKAssertionsAxiom1Drop[Ass]returns the list \
of assertions excluding the trival assertions, according to the axiom 1 of \
the entailment concept."

QEKStar::usage = "QEKStar[PosAss,A] returns the A-star set of the premise A given the positive inferences PosAss."

QEKEquivalentSubsetsQ::usage="QEKEquivalentSubsetsQ[PosAss,A,B] returns True \
iff the subsets A and B are equivalent with respect to the positive \
inferences PosAss."
```

```
QEKInitializeFirstBlock::usage="QEKInitializeFirstBlock[Q]
returns the data structure {OQ,P,N}, representing the first block in the \
Koppens algorithm, consisting of open questions OQ, positive assertions P and \
negative assertions N."

QEKInferences14a::usage="QEKInferences14a[PosAss,Ass]returns the positive \
inferences contributed by the inference rule Apx&BPA&AxPy->Bpy when the \
assertion Ass is positively judged (APx) and added to the previous positive \
assertions PosAss."

QEKInferences14b::usage="QEKInferences14b[PosAss,NegAss,Ass]returns the \
negative inferences contributed by the inference rule APx&AxPB&ANy->BNy when \
the assertion APx is added to the previous positive PosAss and negative \
NegAss assertions."

QEKInferences14c::usage="QEKInferences14c[Q,PosAss,NegAss,Ass]returns the \
negative inferences contributed by the inference rule APx&ByPA&BNx->BNy when \
the assertion APx is added to the previous positive PosAss and negative \
NegAss assertions, given the set Q of items."

QEKInferences14d::usage="QEKInferences14d[Q,PosAss,Ass]returns the negative \
inferences contributed by the inference rule ANx&APB&ByPx->BNy if ANx is \
added to the previous negative assertions NegAss, given the set Q of items."

QEKCollectInferencesForNegativeAnswer::usage=\
"QEKCollectInferencesForNegativeAnswer[Q,PosAss,Ass]returns the data \
structure {P,N} consisting of positive inferences P and negative inferences N \
when a negative answer on the assertion Ass is collected. The set Q of items, \
lists PosAss and NegAss of established positive and negative assertions are \
given in advance. The inference rule is implemented according to Koppen \
(1993),(14d)."

QEKCollectInferencesForPositiveAnswer::usage=\
"QEKCollectInferencesForPositiveAnswer[Q,PosAss,NegAss,Ass]returns the data \
structure {P,N} consisting of positive inferences P and negative inferences N \
when a positive answer on the assertion Ass is collected. The set Q of items, \
lists PosAss and NegAss of established positive and negative assertions are \
given in advance. The inference rule is implemented according to Koppen \
(1993),(14a),(14b),(14c)."

QEKCountInferences::usage="QEKCountInferences[Q,PosAss,NegAss,Ass]returns \
the data structure {Neg,Pos} containg the number Neg of inferences that would \
be drawn in case a negative answer is collected on the assertion Ass, and the \
number Pos of inferences in case of a positive answer. The set Q of items, \
the established positive inferences PosAss and negative inferences NegAss are \
given in advance."

QEKSelectionRuleWeightedSums::usage="QEKSelectionRuleWeightedSums[Q,PosAss,\
NegAss,OpenQ,options] returns the assertion which implies a maximal number of \
inferences according to the weighted sums selection rule. The option \
ProbabilityYES is supported having 0.5 as the default value."

QEKSelectionRuleMaximin::usage="QEKSelectionRuleMaximin[Q,PosAss,NegAss,OpenQ]\
 returns the assertion which implies a maximal number of inferences according \
to the maximin selection rule."

QEKSelectNextQuestion::usage="QEKSelectNextQuestion[Q,Pos,Neg,OpenQ,options]\
returns the data structure {NextAss,NegAnsInf,PosAnsInf} where NextAss is the \
next assertion for querying accompanied with its (both positive and negative) \
inferences NegAnsInf (in case of a negative answer) and PosAnsInf (in case of \
```

```
a positive answer). The selection is conducted on the set Q of items, \
assuming previously collected positive inferences Pos, negative inferences \
Neg, and the items OpenQ left open. The selection rule is given by the option \
SelectionRule that accepts the following values: Maximin (default), \
WeightedSums."

QEKOutput::usage="QEKOutput[Q,PosAss,NegAss,options] returns the output of a \
querying proces either as a knowledge space or as assertions depending on the \
OutputStructure option. The querying process is captured by the positive \
assertions PosAss and the negative assertions NegAss, given the set Q of \
items. The OutputStructure option accepts the following values: \
SpaceStructure (default) and Assertions."

QEKAllAssertionsToSpace::usage="QEKAllAssertionsToSpace[Q,PosAss]returns the \
knowledge space derived from the positive assertions PosAss."

QEKQueryExpertByKoppen::usage="QEKQueryExpertByKoppen[Q,ItOrSkOrExp,options]\
queries an expert using
  Koppen's (1993) block-by-block  algorithm. A human (ItOrSkOrExp represents \
a description of either items or skills) or a simulated expert (ItOrSkOrExp \
represents
  the simulated expert) is queried on the set Q of items, and a structure
  (either a space or assertions) is returned. The valid options are:
   Expert (Human [default] or Simulation), OutputStructure (SpaceStructure
  [default] or Assertions), Verbosity (Off [default] or 1,2),
  QueryOn (KnowledgeSpace [default] or CompetenceSpace), and QSRReport
  (Result [default], Report or Both)."



QEOSQueryTextsInterpretations::usage="QEOSQueryTextsInterpretations[NofSkils,\
TextPart] returns the textual part of a query on interpretation given the \
text part TextPart (introduction, middle, ending) and the number NofSkills \
(1,2, generic, all_minus_one) of skills."

QEOSFormulateQueryInterpretations::usage="QEOSFormulateQueryInterpretations[a,\
Items,C,K,Skills] returns the query question on the given item a described in \
Items given the competence state C of the competence structure K described in \
Skills."

QEOSQueryTextsSkills::usage="QEOSQueryTextsSkills[PremiseSize,TextPart]\
returns the textual part of a query on interpretation given the text part \
TextPart (introduction or question) and the premise size PremiseSize \
(1,2,generic,all_minus_one)."

QEOSFormulateQuerySkills::usage="QEOSFormulateQuerySkills[E,Skills,Assertion]\
returns the query question based on the given assertion Assertion given the \
set E of (elementary) skills described in Skills."

QEOSQueryExpertOnCompetenceSpace::usage="QEOSQueryExpertOnCompetenceSpace[E,\
SkOrExp,options] queries a human
  (SkOrExp represents a description of skills) or a simulated expert (SkOrExp \

  represents a simulated expert) on structure of  the set E of elementary
  competencies and returns the established structure either as a space or \
assertions. An valid option is
  QueryingAlgorithm (Koppen [default]). For additional valid options
  see QEKQueryExpertByKoppen."

MakeIAssertion::usage="MakeIAssertion[CS,Item]returns the interpretation \
```

```
assertion consisting of the competence state CS and the item Item."

IAssertionItem::usage="IAssertionItem[IAss]returns the item of the \
interpretation assertion IAss."

IAssertionCompetenceState::usage="IAssertionCompetenceState[IAss]returns the \
competence state of the interpretation assertion IAss."

QEOSAskExpertOnInterpretation::usage="QEOSAskExpertOnInterpretation[K,Skills,\
Items,IAss,Human] returns an answer on the interpretation assertion IAss from \
a human expert given the competence structure K described in Skills and the \
interpretation assertion item desribed in Items.\n\n
QEOSAskExpertOnInterpretation[IAss,Expert,Simulation] returns an answer on \
the interpretation assertion IAss from the simulated expert Expert."

QEOSInterpretationsOnItem::usage="QEOSInterpretationsOnItem[IASSes,q]returns \
all interpretation assertions given the item q and the list IASSes of \
interpretation assertions."


QEOSInterpretationFunctionCollectInferencesForPositiveAnswer::usage=\
"QEOSInterpretationFunctionCollectInferencesForPositiveAnswer[OpenQ,IAss] \
returns all interpretation assertion inferences given the list OpenQ
  of open interpretation assertions assuming the interpretation assertion \
IAss was judged positively."


QEOSInterpretationFunctionCollectInferencesForNegativeAnswer::usage=\
"QEOSInterpretationFunctionCollectInferencesForNegativeAnswer[OpenQ,IAss] \
returns all interpretation assertion inferences given the list OpenQ
  of open interpretation assertions assuming the interpretation assertion \
IAss was judged negatively."

QEOSInterpretationFunctionCountInferences::usage=\
"QEOSInterpretationFunctionCountInferences[IASSes,IAss]returns the data \
structure {Neg,Pos} containing the number Neg of inferences drawn if a \
negative answer on the interpretation assertion IAss has been collected, and \
the number Pos of inferences drawn in case a positive answer has been \
collected. The list IASSes of open interpretation assertions is given in \
advance."

QEOSInterpretationFunctionSelectionRuleMaximin::usage=\
"QEOSInterpretationFunctionSelectionRuleMaximin[OpenQ] returns the \
interpretation assertion which implies a maximal number of inferences \
according to the maximin selection rule. The list OpenQ of open \
interpretation assertions is given."

QEOSInterpretationFunctionSelectionRuleWeightedSums::usage=\
"QEOSInterpretationFunctionSelectionRuleWeightedSums[OpenQ] returns the \
interpretation assertion which implies a maximal number of inferences \
according to the weighted sums selection rule. The list OpenQ of open \
interpretation assertions is given."

QEOSSelectNextInterpretationQuestion::usage=\
"QEOSSelectNextInterpretationQuestion[OpenQ,opts] returns the next \
interpretation assertion for querying given the (sorted) list OpenQ of open \
interpretation assertions respecting the given options. The valid options \
are: OpenQuestions (Sorted [default],Unsorted) - in case the list OpenQ is \
not sorted, it will be sorted with respect to the cardinality of competence \
states; RuleScope (Block [default], All) - whether to search for the next \
```

```
interpretation assertion only in the next block (defined by the cardinality \
of competence states) or to search all interpretation assertions in OpenQ; \
SelectionRule (Maximin [default], WeightedSums)- which selection rule is to \
be used."


QEOSQueryExpertOnInterpretationFunction::usage=\
"QEOSQueryExpertOnInterpretationFunction[K,A,DescOrExp,opts] establishes an
  interpretation function by querying an expert given the set A
  of items and the competence space K. After the querying only the \
competence-based items set, together with its interpretations, is kept. In \
case of a human expert, DescOrExp
  has the form {Skills,Items}, where the used skills are described in Skills \
and the items in Items. In case of a simulated expert, DescOrExp
  describes the expert. The valid options are: Verbosity (Off [Default],
  1,5,10,15), QSRReport (Result [default], Report or Both), Expert (Human \
[default], Simulation), SelectionRule (Maximin
  [default], WeightedSums) and RuleScope (Block [default], All)."

QEOSInterpretationsFromIAssertions::usage="QEOSInterpretationsFromIAssertions[\
IASSes] returns the list of interpretations derived from the list IASSes of \
interpretation assertions."

QEOSQueryExpertOnDiagnostic::usage="
  QEOSQueryExpertOnDiagnostic[E,A,DescOrExp,opts] queries a human expert \
(DescOrExp having the form {Skills,Items} describes skills and items) or a \
simulated expert (DescOrExp represents a simulated expert whose
  metaknowledge is given as a diagnostic) on competence-performance
  diagnostic.  The set A of items and the set E of elementary
  competencies are given. The valid options are: Expert (Human [default] or
  Simulation),  QSRReport (Result [default], Report or
  Both). For other valid options see: QEOSQueryExpertOnCompetenceSpace
  and  QEOSQueryExpertOnInterpretationFunction."



QSRMakeSonde::usage="QSRMakeSonde[OQB,NQ] returns a querying session \
observation sonde consisting
  of the number OQ of open questions at the start of the observation and the \
number NQ of questions presented to an expert during the observation.

QSRSondeOpenQuestionsInitial::usage="QSRSondeOpenQuestionsInitial[Sonde] \
returns the initial number of open
  questions recorded by the querying session Sonde."

QSRSondeQuestionsAsked::usage="QSRSondeQuestionsAsked[Sonde] returns the \
number of questions presented to an expert recorded by the querying session \
sonde Sonde."

QSRSondeOpenQuestionsInitialReplace::usage=\
"QSRSondeOpenQuestionsInitialReplace[Sonde,N] returns the querying
  session Sonde after replacing the initial numberof open questions by N."

QSRSondeOpenQuestionsInitialAdd::usage="QSRSondeOpenQuestionsInitialAdd[Sonde,\
N] returns the querying session Sonde after incrementing the initial number \
of open questions by N."

QSRSondeQuestionsAskedReplace::usage="QSRSondeQuestionsAskedReplace[Sonde,N] \
returns the querying session sonde
  Sonde after replacing the number of asked questions by N."
```

```
QSRSondeQuestionsAskedAdd::usage="QSRSondeQuestionsAskedAdd[Sonde,N] returns \
the querying session sonde
  Sonde after incrementing the number of presented questions by N."



QEQueryExpertOnKnowledgeSpace::usage="QEQueryExpertOnKnowledgeSpace[Q,ItOrExp,\
options] queries a human
  (ItOrExp takes a description of items) or a simulated expert (ItOrExp
  represents a simulated expert) on structure of the set Q of items. The \
established structure is returned either as a space or assertions. A valid \
option is
  QueryingAlgorithm (Koppen [default]). For additional valid options
  see QEKQueryExpertByKoppen."




Expert::usage="Expert is an option of QEKQueryExpertByKoppen, \
QEOSQueryExpertOnDiagnostic and QEOSQueryExpertOnInterpretationFunction. The \
possible values are Human (default) and Simulation.";

OpenQuestions::usage="OpenQuestions is an option of \
QEOSSelectNextInterpretationQuestion. The possible values are Sorted \
(default) and Unsorted.";

OutputStructure::usage="OutputStructure is an option of QEKOutput, \
QEKQueryExpertByKoppen. The possible values are SpaceStructure (default) and \
Assertions.";

ProbabilityYES::usage="ProbabilityYES is an option of \
QEKSelectionRuleWeightedSums and \
QEOSInterpretationFunctionSelectionRuleWeightedSums. The default value is \
0.5.";

QueryingAlgorithm::usage="Querying algorithm is an option of \
QEOSQueryExpertOnCompetenceState and QEQueryExpertOnKnowledgeSpace. The \
possible value is Koppen (default).";

QueryOn::usage="QueryOn is an option of QEKAskExpert, QEKQueryExpertByKoppen. \
The possible values are KnowledgeSpace (default) and CompetenceSpace.";

QSRReport::usage="QSRReport is an option of QEKQueryExpertByKoppen, \
QEOSQueryExpertOnDiagnostic and QEOSQueryExpertOnInterpretationFunction. The \
possible values are Result (default), Report and Both.";

SelectionRule::usage="SelectionRule is an option of QEKSelectNextQuestion, \
QEOSSelectNextInterpretationQuestion and \
QEOSQueryExpertOnInterpretationFunction. The possible values are Maximin \
(default) and WeightedSums.";

RuleScope::usage="RuleScope is an option of \
QEOSSelectNextInterpretationQuestion and \
QEOSQueryExpertOnInterpretationFunction. The possible values are Block \
(default) and All.";
```

```
Options[QEQueryExpertOnKnowledgeSpace]={QueryingAlgorithm->Koppen};

Options[QEKAskExpert]={QueryOn->KnowledgeSpace};

Options[QEKQueryExpertByKoppen]={Verbosity:>$Verbosity,Expert->Human,
   OutputStructure->SpaceStructure,
   QueryOn->KnowledgeSpace,QSRReport->Result};

Options[QEKSelectionRuleWeightedSums]={ProbabilityYES->0.5};

Options[QEKSelectNextQuestion]={SelectionRule->Maximin};

Options[QEKOutput]:={ OutputStructure->SpaceStructure};

Options[QEOSInterpretationFunctionSelectionRuleWeightedSums]={ProbabilityYES->\
.5};

Options[QEOSSelectNextInterpretationQuestion]={OpenQuestions->Sorted,
   RuleScope->Block, SelectionRule->Maximin};

Options[QEOSQueryExpertOnCompetenceSpace]={QueryingAlgorithm->Koppen};

Options[QEOSQueryExpertOnDiagnostic]={Expert->Human,QSRReport->Result};

Options[QEOSQueryExpertOnInterpretationFunction]={Verbosity:>$Verbosity,
   Expert->Human,RuleScope->Block,
   SelectionRule->Maximin,QSRReport->Result};




Begin["`Private`"]


(* QEKAssertionPremise[Assertion] returns the premise of the Assertion *)
QEKAssertionPremise[Assertion_]:=Assertion[[1]]

(* QEKAssertionConsequence[
     Assertion] returns the consequence of the Assertion *)
QEKAssertionConsequence[Assertion_]:=Assertion[[2]]

(* QEKMakeAssertions[Q,
     Premise] returns the list of all possible assertions for the given \
Premise and the set Q of items,
  under the  assumtion of only one element in the consequence of the \
assertion,  *)
QEKMakeAssertions[Q_,Premise_]:=Module[{},
   Map[({Premise,#})&,Q]
   ]

(* QEKMakeFullBlock[Q,k] returns the block of the subsets-by-
   items table with the size of the premise k for the set Q of items *)
QEKMakeFullBlock[Q_,k_]:=
  Module[{},
   Flatten[Map[Function[A,QEKMakeAssertions[Q,A]],
      KSubsets[Q,k]],1]
   ]

(* QEKMakeSbITable[Q] returns the subsets-by-
   items table for the given set Q of items *)
```

```
QEKMakeSbITable[Q_]:=Module[{k,n:=SetCardinality[Q]},
   Flatten[Table[QEKMakeFullBlock[Q,k],{k,1,n}],1]
   ]

(* QEKQueryTexts[PremiseSize,
     TextPart] returns the TextPart (introduction or question) for the \
premise of the size PremiseSize (1,2,generic,previous_to_last) *)
QEKQueryTexts[PremiseSize_,TextPart0_]:=
 Module[{QueryTexts,TextParts,TextPart},
   TextParts={{introduction,2},{question,3}};
   QueryTexts={{1,
       "Suppose that a student under examination has just failed the \
problem ",
       " Is it then practically certain that this student will also fail \
the problem "},{2,
       "Suppose that a student under examination has just failed both the \
problems ",
       " Is it then practically certain that this student will also fail \
the problem "},{generic,
       "Suppose that a student under examination has just failed all the \
problems: ",
       " Is it then practically certain that this student will also fail \
the problem "},{previous_to_last,
       "Suppose you are tutoring a student who does not master any of the \
problems in the given problem set.",
       " Could you then start by teaching him how to solve the problem \
"}};
   TextPart:=Select[TextParts,(#[[1]]===TextPart0)&,1][[1,2]];
   Return[Select[QueryTexts,(#[[1]]===PremiseSize)&,1][[1,TextPart]]]
   ]

(* QEKFormulateQuery[Q,Items,
     Assertion] returns the query question on the given assertion Assertion \
formulated for a human expert, given the set Q of items described in Items *)


QEKFormulateQuery[Q_,Items_,Assertion_]:=
 Module[{nQ:=SetCardinality[Q],
    nAssertion:=SetCardinality[QEKAssertionPremise[Assertion]],
    A:=QEKAssertionPremise[Assertion],b:=QEKAssertionConsequence[Assertion],
    QueryTexts,QueryText,Text1,Text2,Text3,Text4,tmp={},Ntmp},
   Which[
    nAssertion==1,
    Block[{},
     Text1:= QEKQueryTexts[1,introduction];
     Text2:=ItemDescription[ItemGet[Items,A[[1]]]];
      tmp= QEKQueryTexts[1,question];
     Text3= Apply[StringJoin,{tmp,ItemDescription[ItemGet[Items,b]],"?"}];
     QueryText:=StringJoin[{Text1,Text2,".",Text3}];
     ]
    ,nAssertion==2,
    Block[{},
     Text1= QEKQueryTexts[2,introduction];
     Text2=
      StringJoin[{ItemDescription[ItemGet[Items,A[[1]]]]," and ",
          ItemDescription[ItemGet[Items,A[[2]]]],"."}];
     tmp= QEKQueryTexts[2,question];
     Text3=Apply[StringJoin,{tmp,ItemDescription[ItemGet[Items,b]],"?"}];
     QueryText:=StringJoin[{Text1,Text2,Text3}];
     ]
```

```
    ,(nQ-nAssertion)==1,
    Block[{},
     Text1= QEKQueryTexts[previous_to_last,introduction];
     Text2="";
     tmp= QEKQueryTexts[previous_to_last,question];
     Text3=Apply[StringJoin,{tmp,ItemDescription[ItemGet[Items,b]],"?"}];
     QueryText:=StringJoin[{Text1,Text2,Text3}];
     ]
    ,True (* GENERIC *),
    Block[{},
     Text1= QEKQueryTexts[generic,introduction];
     tmp=Map[(ItemDescription[ItemGet[Items,#]])&,A];
     Ntmp=Length[tmp];
     Text2=Apply[StringJoin,Table[tmp[[i]]<>",",{i,1,Ntmp-1}]];
     Text2=Apply[StringJoin,{Text2,tmp[[Ntmp]],"."}];
     tmp= QEKQueryTexts[generic,question];
     Text3=Apply[StringJoin,{tmp,ItemDescription[ItemGet[Items,b]],"?"}];
     QueryText:=StringJoin[{Text1,Text2,Text3}];
     ]
    ];
  Text4=
   " You may assume that chance factors, such as careless errors and lucky \
guesses, play no rule in the student's performance.";
  Return[StringJoin[{QueryText,Text4}]];
  ]

(* QEKAskExpert[QorE,ItemsOrSkills,Assertion,Human,
    options] returns an answer from a human experts on assertion containing \
some items from QorE interpreted in ItemsOrSkills in case of querying on \
knowledge space,
 or some skills from QorE interpreted in ItemsOrSkills in case of querying \
on competence space. Available options are:
    QueryOn ( KnowledgeSpace [default]
          or CompetenceSpace). *)
QEKAskExpert[Q_List?SetQ,Items_List,Assertion_,Human,opts___]:=
 Module[{answer,CertaintyFactor,OptionQueryOn},

  (* Querying on knowledge space or competence space? *)
  OptionQueryOn = QueryOn /. {opts} /. Options[QEKAskExpert];

  (* Formulate a query on items *)
  If[OptionQueryOn==KnowledgeSpace,
   answer=Input[QEKFormulateQuery[Q,Items,Assertion]];
   ];

  (* Formulate a query on skills *)
  If[OptionQueryOn==CompetenceSpace,
   answer=Input[QEOSFormulateQuerySkills[Q,Items,Assertion]];
   ];


  CertaintyFactor=1;
  If[MemberQ[{"Y","T","YES","TRUE"},ToUpperCase[ToString[answer]]],
   answer=True,answer=False];
  Return[ExpertMakeAnswer[Assertion,answer,CertaintyFactor]]
  ]

(* QEKAskExpert[Assert,Exp,
    Simulation] returns an answer from the expert Exp on assertion \
Assert. Expert's meta-
```

```
   knowledge may be represented either with a structure or a diagnostic. *)
QEKAskExpert[Assertion_,Expert_,Simulation]:=
 Module[{answer,ExpMK,ExpMKAssPos,ExpCErrors},

   (* Extract the expert's meta-knowledge *)
   ExpMK=ExpertMetaKnowledge[Expert];

   (* Check the type of the stored meta-knowledge,
    and convert it to assertions
     if necessary *)
   If[Head[ExpMK]==Diagnostic,
    ExpMK=MakeMetaKnowledge[
      EntailmentFromKnowledgeSpace[
       Structure[CPADiagnosticElementaryCompetencies[ExpMK],
        CPADiagnosticCompetenceSpace[ExpMK]]
       ],
      Assertions];];
   If[Head[ExpMK]==Structure,
    ExpMK=
     MakeMetaKnowledge[EntailmentFromKnowledgeSpace[ExpMK],Assertions];
    ];

   ExpMKAssPos=MetaKnowledgeAssertionsPositive[ExpMK];
   ExpCErrors=ExpertCarelessErrors[Expert];
   answer=ExpertMakeAnswer[Assertion,MemberQ[ExpMKAssPos,Assertion],1];
   Return[answer]
   ]


(* QEKExtractPositiveJudgments[
    QH] returns extracted positive judgments from the query history QH *)
QEKExtractPositiveJudgmentsFromQueryHistory[QueryHistory_]:=Module[{},
   Map[(ExpertAnswerAssertion[#])&
    ,Select[QueryHistory,
      Function[answer,ExpertAnswerLogicalValue[answer]==True]
      ]] ]

(* QEKExtractNegativeJudgments[
    QH] returns extracted positive judgments from the query history QH *)
QEKExtractNegativeJudgmentsFromQueryHistory[QueryHistory_]:=Module[{},
   Map[(ExpertAnswerAssertion[#])&
    ,Select[QueryHistory,
      Function[answer,ExpertAnswerLogicalValue[answer]==False]
      ]] ]

(* QEKQueryExpertStraightforward[Q,Items,
    Human] returns the history of jugdments obtained while querying the \
human expert for structure on items Q interpreted in Items *)
QEKQueryExpertStraightforward[Q_,Items_,Human]:=
 Module[{QueryHistory,OpenQuestions,PJudgments,TrivialJudgments},
   (* Exclude trivial assertions from the open questions *)
   OpenQuestions=QEKAssertionsAxiom1Drop[PQxQ[Q]];
   QueryHistory=
    Function[Ass,QEKAskExpert[Q,Items,Ass,Human]]/@ OpenQuestions;
   (* Adds trivial assertions to the query history *)
   TrivialJudgments=Map[({#,True,"I"})&,QEKAssertionsAxiom1[Q]];
   (* Returns the query history including trivial judgements*)
   Return[Join[QueryHistory,TrivialJudgments]];
   ]
```

```
(* QEKQueryExpertStraightforward[Q,Expert,
     Simulation] returns the history of jugdmenets obtained while querying \
the simulated Expert for structure on items Q *)
QEKQueryExpertStraightforward[Q_,Expert_,Simulation]:=
 Module[{QueryHistory,OpenQuestions,PJudgments,TrivialJudgments},
  (* Exclude trivial assertions from the open questions *)
  OpenQuestions=QEKAssertionsAxiom1Drop[PQxQ[Q]];
  (* Record all answers from the expert *)
  QueryHistory=
   Map[Function[Ass,QEKAskExpert[Ass,Expert,Simulation]],OpenQuestions];
  TrivialJudgments=Map[({#,True,"I"})&,QEKAssertionsAxiom1[Q]];
  (* Returns the query history including trivial judgements*)
  Return[Join[QueryHistory,TrivialJudgments]];
  ]


(* QEKEntailmentFromQueryHistory[Q,
     QueryHistory] extracts only positive judgments from the query history \
of the establishment of a structure of the items Q *)
QEKEntailmentFromQueryHistory[Q_,QueryHistory_]:=Module[{PJudgments},
  PJudgments=QEKExtractPositiveJudgmentsFromQueryHistory[QueryHistory];
  Return[Relation[Q,PJudgments]]
  ]

(* QEKEstablishKnowledgeSpaceStraightforward[Q,Items,
     Human] returns the structure on the items Q interpreted in Items \
established by querying human Expert *)
QEKEstablishKnowledgeSpaceStraightforward[Q_,Items_,Human]:=
 EntailmentToKnowledgeSpace[
  QEKEntailmentFromQueryHistory[Q,
   QEKQueryExpertStraightforward[Q,Items,Human]
   ]
  ]

(* QEKEstablishKnowledgeSpaceStraightforward[Q,Expert,
     Simulation] returns the structure on the items Q obtained by querying \
simulated Expert *)
QEKEstablishKnowledgeSpaceStraightforward[Q_,Expert_,Simulation]:=
 EntailmentToKnowledgeSpace[
  QEKEntailmentFromQueryHistory[Q,
   QEKQueryExpertStraightforward[Q,Expert,Simulation]
   ]
  ]

(* QEKAssertionsAxiom1Drop[
     Ass] returns the list of assertions excluding the trival assertions \
according to the axiom1 of the entailment *)
QEKAssertionsAxiom1Drop[Assertions_]:=Module[{},
  Select[Assertions,
   Function[Ass,Block[{A=Ass[[1]],q=Ass[[2]]}, ¬SetElementQ[q,A]]]]
  ]

(* QEKAssertionsAxiom1[
     Q] returns the list of all trivial assertions according to the axiom1 \
of the entailment *)
QEKAssertionsAxiom1[Q_]:=Module[{Assertions},
  Assertions=PQxQ[Q];
  Select[Assertions,
   Function[Ass,Block[{A=Ass[[1]],q=Ass[[2]]}, SetElementQ[q,A]]]]
  ]
```

```
(* QEKStar[PosAss,A] returns the A-
   star of the premise A given the positive inferences PosAss  *)
QEKStar[PosAssertions_,A_]:=Module[{},
   Union[Map[(#[[2]])&,
      Select[PosAssertions,(QEKAssertionPremise[#]==A)&]
      ]]
   ]

(* QEKEquivalentSubsetsQ[PosAss,A,
     B] returns True iff the subsets A and B are equivalent with respecto to \
the positive inferences PosAss *)
QEKEquivalentSubsetsQ[PosAssertions_,A_,B_]:=Module[{},
   QEKStar[PosAssertions,A]==QEKStar[PosAssertions,B]
   ]

(* QEKInitializeFirstBlock[Q] returns the data structure {OQ,P,N},
  representing the first block in Koppen's algorithm,
  where OQ represents open questions,
  P positive assertions and N negative assertions  *)
QEKInitializeFirstBlock[Q_]:=Module[
   {OpenQ,PosAss,NegAss},
   NegAss={};
   PosAss=Map[({#}, # })&,Q];
   OpenQ=QEKAssertionsAxiom1Drop[QEKMakeFullBlock[Q,1]];
   Return[{OpenQ,PosAss,NegAss}];
   ]

(* QEKInferences14a[PosAss,
            Ass] returns the positive inferences contributed by the \
inference rule Apx&BPA&AxPy->
   Bpy if APx would be added to the previous positive assertions PosAss *)
QEKInferences14a[PosAss_,Ass_]:=Module[
   {A:=QEKAssertionPremise[Ass], x:=QEKAssertionConsequence[Ass],Ax,Ys,Bs},
   Ax=Union[A,{x}];
   Ys=Union[QEKStar[PosAss,Ax]];
   Bs=Union[
      Map[(QEKAssertionPremise[#])&,
        Select[PosAss,(EqualSetsQ[QEKStar[PosAss,QEKAssertionPremise[#]],
             A])&]
        ]];
   If[(Bs\[NotEqual]{}\[And]Ys\[NotEqual]{}),CartesianProduct[Bs,Ys],{}]
   ]

(* QEKInferences14b[PosAss,NegAss,
            Ass] returns all the negative inferences made by the \
inference rule APx&AxPB&ANy->
   BNy if the assertion Apx would be added to the previous positive PosAss \
and negative NegAss assertions  *)
QEKInferences14b[PosAss_,NegAss_,Ass_]:=Module[
   {A=QEKAssertionPremise[Ass], x=QEKAssertionConsequence[Ass],Ax,Ys,B},
   Ax=Union[A,{x}];
   B=Union[QEKStar[PosAss,Ax]];
   Ys=Union[QEKStar[NegAss,A]];
   If[(B\[NotEqual]{}\[And]Ys\[NotEqual]{}),Map[({B,#})&,Ys],{}]
   ]

(* QEKInferences14c[Q,PosAss,NegAss,
```

```
            Ass] returns all the negative inferences made by the \
inference rule APx&ByPA&BNx->
   BNy if the assertion Apx would be added to the previous positive PosAss \
and negative NegAss assertions, given the set Q of items  *)
QEKInferences14c[Q_,PosAss_,NegAss_,Ass_]:=Module[
   {A=QEKAssertionPremise[Ass] , x=QEKAssertionConsequence[Ass],Bs,BNys,
    tmp, tmp1},
   Bs=Map[(QEKAssertionPremise[#])&,
     Select[NegAss,(QEKAssertionConsequence[#]==x)&]];
   BNys={};
   Do[
    B=Bs[[i]];
    tmp=
     Select[Map[({Union[B,{#}],#})&,
       Q], (EqualSetsQ[QEKStar[PosAss,#[[1]]],A])&];
    tmp1=Union[BNys,tmp];
    BNys=tmp1;
    ,{i,1,Length[Bs]}];
   Return[BNys]
   ]

(* QEKInferences14d[Q,PosAss,
            Ass] returns the negative inferences contributed by the \
inference rule ANx&APB&ByPx->
   BNy  if ANx would be added to the previous negative assertions NegAss,
  given the set Q of items *)
QEKInferences14d[Q_,PosAss_,Ass_]:=Module[
   {A:=QEKAssertionPremise[Ass] , x:=QEKAssertionConsequence[Ass],B},
   B=QEKStar[PosAss,A];
   Select[Map[({Union[B,{#}],#})&,Q],
    (SetElementQ[{#[[1]],x},PosAss])&]
   ]

(* QEKCollectInferencesForNegativeAnswer[Q,PosAss,
     Ass] returns the data structure {P,
     N} consisting of positive inferences P and negative inferences N when \
negative answer on asswertion Ass is given.
     The set Q of items and the list PosAss of established positive \
assertions are given in advance,
  and inference rule implemented according to (Koppen, (14d)) *)
QEKCollectInferencesForNegativeAnswer[Q_,PosAss_,Ass_]:=
  Return[{{},QEKInferences14d[Q,PosAss,Ass]}]


(* QEKCollectInferencesForPositiveAnswer[Q,PosAss,NegAss,
     Ass] returns the datastructure {P,
     N} consisting of positive inferences P and negative inferences N when \
possitive answer on assertion Ass is given. The set Q of items,
  lists PosAss and NegAss of established positive and negative assertions are \
given in advance,
  and inference rule implemented according to (Koppen,(14a),(14b),(14c)) *)
QEKCollectInferencesForPositiveAnswer[Q_,PosAss_,NegAss_,Ass_]:=Module[{},
   {QEKInferences14a[PosAss,Ass],
    Union[QEKInferences14b[PosAss,NegAss,Ass],
     QEKInferences14c[Q,PosAss,NegAss,Ass]]}
   ]


(* QEKCountInferences[Q,PosAss,NegAss,Ass] returns the data structure {Neg,
     Pos} containg the number Neg of inferences that would be drawn in case \
```

```
the negative answer is given on the assertion Ass,
  and the number Pos of inferences in case of positive answer.
     The set Q of items,
  the established positive inferences PosAss and negative inferences NegAss \
are given.  *)
QEKCountInferences[Q_,PosAss_,NegAss_,Ass_]:=Module[{},
   Return[
    {Apply[Plus,
       Map[(Length[#])&,
         QEKCollectInferencesForNegativeAnswer[Q,PosAss,Ass]]],
      Apply[Plus,
       Map[(Length[#])&,
         QEKCollectInferencesForPositiveAnswer[Q,PosAss,NegAss,Ass]]]}]
   ]

(* QEKSelectionRuleWeightedSums[Q,PosAss,NegAss,OpenQ,
     options] returns an assertion which has maximal expected number of \
inferences according to weighted sums selection rule.
     The option ProbabilityYES is supported having 0.5 as the default value. \
*)

QEKSelectionRuleWeightedSums[Q_,PosAss_,NegAss_,OpenQ_,opts___]:=
  Module[{PrYES,PrNO,candidates,MaxAss,i},
   PrYES=ProbabilityYES /. {opts} /. Options[QEKSelectionRuleWeightedSums];
   PrNO=1-PrYES;
   candidates=
    Map[({#[[1]], #[[2,1]]*PrNO+#[[2,2]]*PrYES })&,
      Map[({#,QEKCountInferences[Q,PosAss,NegAss,#]})&,OpenQ]] ;
   MaxAss=candidates[[1]];
   Do[
    If[candidates[[i,2]]>MaxAss[[2]],MaxAss=candidates[[i]]];
    ,{i,2,Length[candidates]}];
   Return[MaxAss[[1]]]
   ]

(* QEKSelectionRuleMaximin[Q,PosAss,NegAss,
     OpenQ] returns an assertion which has maximal expected number of \
inferences according to maximin selection rule. *)
QEKSelectionRuleMaximin[Q_,PosAss_,NegAss_,OpenQ_]:=
  Module[{candidates,MaxAss,i},
   candidates=
    Map[({#[[1]], Min[#[[2,1]],#[[2,2]]]  })&,
      Map[({#,QEKCountInferences[Q,PosAss,NegAss,#]})&,OpenQ]] ;
   tmp:=Map[({#,QEKCountInferences[Q,PosAss,NegAss,#]})&,OpenQ];
   MaxAss=candidates[[1]];
   Do[
    If[candidates[[i,2]]>MaxAss[[2]],MaxAss=candidates[[i]]];
    ,{i,2,Length[candidates]}];
   Return[MaxAss[[1]]]
   ]

(* QEKSelectNextQuestion[Q,Pos,Neg,OpenQ,
     options] returns the data structure {NextAss,NegAnsInf,
     PosAnsInf} where NextAss is a next assertion for querying accompanied \
with its (both positive and negative) inferences NegAnsInf (in case of a \
negative answer) and PosAnsInf (in case of a positive answer).
     Querying is conducted on set Q of items,
  assuming previously established positive inferences Pos,
  negative inferences Neg, open questions OpenQ,
  and it is based on the selection rule defined by the option SelectionRule \
```

```
that accepts the following values:  Maximin (default), WeightedSums. *)

QEKSelectNextQuestion[Q_,PosAss_,NegAss_,OpenQ_,opts___]:=
 Module[{rule,NextAss},
  rule:=SelectionRule /. {opts} /. Options[QEKSelectNextQuestion];
  Switch[rule,
   WeightedSums,
   NextAss=QEKSelectionRuleWeightedSums[Q,PosAss,NegAss,OpenQ,
     SelectionRule->WeightedSums],
   _,NextAss=QEKSelectionRuleMaximin[Q,PosAss,NegAss,OpenQ]
   ];
  Return[{NextAss,QEKCollectInferencesForNegativeAnswer[Q,PosAss,NextAss],
    QEKCollectInferencesForPositiveAnswer[Q,PosAss,NegAss,NextAss]}];
  ]

(* QEKOutput[Q,PosAss,NegAss,
    options] returns the output of the querying proces either as knowledge \
space or as assertions dependingly on the OutputStructure option.The \
OutputStructure option accepts the following values:
    SpaceStructure (default), Assertions. *)

QEKOutput[Q_,PosAss_,NegAss_,opts___]:=Module[{choice},
  choice=OutputStructure  /. {opts} /.Options[QEKOutput];
  Switch[choice,
   Assertions,{PosAss,NegAss,{}},
   _,QEKAllAssertionsToSpace[Q,PosAss]]
  ]

(* QEKAllAssertionsToSpace[Q,
    PosAss] returns the knowledge space established by positive assertions \
PosAss obtained through completed querying. *)
QEKAllAssertionsToSpace[Q_,PosAss_]:=Module[{F},
  F=Union[Map[(SetComplement[Q,QEKStar[PosAss,#]])&,PowerSet[Q]]];
  Return[Structure[Q,F]]
  ]

(* QEKQueryExpertByKoppen[Q,ItOrSkOrExp,options]queries an expert using
  Koppen's (1993) block  algorithm. A human (ItOrSkOrExp represents
    interpretation of items or skills) or a simulated expert (ItOrSkOrExp \
represents
    the simulated expert) is queried on the set Q of items,
 and a structure
  (either space or assertions) is returned. The acceptable options are:
    Expert (Human [default] or Simulation),
 OutputStructure (SpaceStructure
     [default] or Assertions), Verbosity (Off [default] or 1,2),
 QueryOn (KnowledgeSpace [default] or CompetenceSpace), and QSRReport
  (Result [default], Report or Both).  *)

QEKQueryExpertByKoppen[Q_,ItemsExpert_,opts___]:=Module[
  {OpenQ,PosAss,NegAss,Next,NextAss,Answer,NegAnsInf,PosAnsInf,tmp,
   kBlockAs,k,A,B,Xs,classes,NewNegAss,NewPosAss,NofQuestions,QSR,
   OptionVerbosity,OptionHumOrSim,OptionOutputStructure,
    OptionQueryOn,OptionQSRReport},

  (* Whether to query expert on a knowledge space or a competence space *)
  OptionQueryOn:=QueryOn /. {opts} /. Options[QEKQueryExpertByKoppen];


  (* Whether to query human or simulated expert *)
```

```
  OptionHumOrSim:=Expert /. {opts} /. Options[QEKQueryExpertByKoppen];

  (* The kind of output structure *)
  OptionOutputStructure:=OutputStructure /. {opts}
    /. Options[QEKQueryExpertByKoppen];

  (* What the function should return as its output? *)
  OptionQSRReport:=QSRReport /. {opts} /. Options[QEKQueryExpertByKoppen];

  (* Verbal report on querying or not? *)
  OptionVerbosity=Verbosity /. {opts} /. Options[QEKQueryExpertByKoppen];

  If[OptionVerbosity>=1,
    Print[
     "QEOS v1.0, Querying Session Report.\nKoppen (1993) block by block \
algorithm used for querying expert (",OptionHumOrSim,") on ",
      OptionQueryOn, "."];
    Print["Function returns (Result, Report or Both): ",OptionQSRReport];
    Print["Resulting structure: ",OptionOutputStructure,"."];
    ];

  (* VERBOSITY *)
  If[OptionHumOrSim==Human && OptionQueryOn==KnowledgeSpace
     &&OptionVerbosity>=5,
    Print["Description of used items:\n",ItemsExpert];
    ];
  If[OptionHumOrSim==Human && OptionQueryOn==CompetenceSpace
     &&OptionVerbosity>=5,
    Print["Description of used skills:\n",ItemsExpert];
    ];
  If[(OptionHumOrSim==Simulation) && (OptionVerbosity>=5),
    Print["Simulated expert:\n ",ItemsExpert];
    ];




  (* Construct the 1st block and initialize it  *)
  (* premise size *)
   k=1;
  {OpenQ,PosAss,NegAss}=QEKInitializeFirstBlock[Q];

  (* Number of questions asked *)
  NofQuestions=0;

  (* SONDE *)
  (* Create and initialize a querying session sonde if requested *)
  If[OptionQSRReport==Report \[Or] OptionQSRReport==Both,
    QSR=QSRMakeSonde[0,0];
    QSR=QSRSondeOpenQuestionsInitialReplace[QSR,Length[OpenQ]];
    QSR= QSRSondeQuestionsAskedReplace[QSR,NofQuestions];
    ];

  (* VERBOSITY *) (* SONDE *)
  If[OptionVerbosity>=10 &&
      (OptionQSRReport==Report|| OptionQSRReport==Both) ,
    Print["Initial querying session sonde status: ",QSR];
    ];
```

```
(* While a new constructed block of questions is not empty *)
While[OpenQ!={},

 (* VERBOSITY *)
 If[OptionVerbosity>=5,
  Print["New block of questions constructed.\nOpen questions:\n",
   OpenQ];
  ];

 (* VERBOSITY *)
 If[OptionVerbosity>=5,
  Print["Known positive assertions:\n",PosAss];
  Print["Known negative assertions:\n",NegAss];
  ];

 (* Premises of the open questions belonging to the current block *)
 kBlockAs=Map[(#[[1]])&,OpenQ];

 (* Fill the block in with both inferences and answers *)
 (* Any open questions in the current block? *)
 While[OpenQ\[NotEqual]{},


  (* Get the next question,
   store possible inferences using the following
    inference rules: 14a, 14b,14c, 14d (Koppen,1993) *)
  Next=QEKSelectNextQuestion[Q,PosAss,NegAss,OpenQ];
  NextAss=Next[[1]]; NegAnsInf=Next[[2]]; PosAnsInf=Next[[3]];


  (* Query a human or simulated expert for the answer on either items
      or skills  *)
  If[OptionHumOrSim==Simulation,
   Answer=QEKAskExpert[NextAss,ItemsExpert,Simulation];
   ];
  If[OptionHumOrSim==Human && OptionQueryOn==KnowledgeSpace,
   Answer=QEKAskExpert[Q,ItemsExpert,NextAss,Human,
    QueryOn->KnowledgeSpace];];
  If[OptionHumOrSim==Human && OptionQueryOn==CompetenceSpace,
   Answer=QEKAskExpert[Q,ItemsExpert,NextAss,Human,
    QueryOn->CompetenceSpace];];


  NofQuestions++;


  (* VERBOSITY *)
  If[OptionVerbosity>=5,
   Print[NofQuestions,". question: ",NextAss,". The answer: ",Answer];
   ];



  (* Add the answer and the appropriate inferences to the
     established assertions *)
  If[ExpertAnswerLogicalValue[Answer],
   AppendTo[PosAss,ExpertAnswerAssertion[Answer]];
   If[PosAnsInf[[1]]\[NotEqual]{},AppendTo[PosAss,PosAnsInf[[1]]]];
   If[PosAnsInf[[2]]\[NotEqual]{},AppendTo[NegAss,PosAnsInf[[2]]]];
```

```
    ,AppendTo[NegAss,ExpertAnswerAssertion[Answer]];
    If[NegAnsInf[[1]]\[NotEqual]{},AppendTo[PosAss,NegAnsInf[[1]]]];
    If[NegAnsInf[[2]]\[NotEqual]{},
     AppendTo[NegAss,NegAnsInf[[2]]]];    ];

  (* Update the list of open questions *)
  tmp=SetDifference[OpenQ,{NextAss}];OpenQ=tmp;

  (* VERBOSITY *)
  If[OptionVerbosity>=10,
   Print["Block filled with anwers and inferences."];
   Print["Known positive assertions:\n",PosAss];
   Print["Known negative assertions:\n",NegAss];];

  ];  (* End of While[] *)


(* Construct the next block of  open questions by taking only \
appropriate candidates *)

  (* Increase the premise size *)
  k++;

  (* First step, take only appropriate candidates from the previous
     block. (Koppen 1993, S4.2,[1])  *)
  candidates={};
  Do[
   A=kBlockAs[[i]];
   Xs=SetComplement[Q,QEKStar[PosAss,A]];
   tmp=Join[candidates,Map[(Union[A,{#}])&,Xs]];
   candidates=Union[tmp];
   ,{i,1,Length[kBlockAs]}];

  (* VERBOSITY *)
  If[OptionVerbosity>=10,
   Print["Constructing the next block of questions.\n"];
   Print["1. step premise-candidates:\n",candidates];];

  (* Second step, take only one representative of each equivalent class.
      (Koppen 1993, S4.2,[2]) *)
  classes={};
  Do[
   A=candidates[[i]];
   tmp=Select[candidates,(QEKEquivalentSubsetsQ[PosAss,A,#])&] ;
   classes=Join[classes,{tmp}];
   ,{i,1,Length[candidates]}];

  (* VERBOSITY *)
  If[OptionVerbosity>=15,
   Print["Equivalence classes of premise-candidates:\n",classes];
   ];

  (* Collect temporary candidates *)
  candidates=Map[(#[[1]])&,Union[classes]];

  (* VERBOSITY *)
  If[OptionVerbosity>=10,
   Print["2. step premise-candidates:\n",candidates];];
```

```
(* Third step,
  only candidates Bs for which (A is subset of B and B is subset of A-
      star)does not hold. (Koppen 1993, S4.2,[3])  *)
kBlockAs={};
Do[
 B=candidates[[i]];flag=True;
 Do[
  A=PosAss[j];AStar=QEKStar[PosAss,A];
  If[SubsetQ[A,B]\[And]SubsetQ[B,Astar],flag=False;Break[];];
  ,{j,1,Length[PosAss]}];
 If[flag,kBlockAs=Join[OpenQ,{B}]];

 (* VERBOSITY
    If[OptionVerbosity>=15,Print["kBlockAs: ",kBlockAs];]; *)

 ,{i,1,Length[candidates]}];


(* Construct temporary new block with open questions *)
OpenQ=Flatten[Map[Function[A,Map[({A,#})&,Q]],kBlockAs],1];

(* VERBOSITY *)
If[OptionVerbosity>=10,
 Print["3. step candidates:\n",OpenQ];
 ];

(* VERBOSITY *)
If[OptionVerbosity>=10,
 Print["Filling in the new block with already known inferences."]];

(* Filling in the new block with already known inferences *)
AllBs=Apply[Union,Table[KSubsets[Q,i],{i,1,(k-1)}]];
NewPosAss=NewNegAss={};

(* Take As, one by one, from the newly constructed block *)
Scan[Function[A,

  (* First, positive inferences: Calculating A-plus for a given A *)
  flag=True;
  Aplus=OpenQ[[1,1]];
  While[flag,
   i=0;
   While[flag,
    B=QEKAssertionPremise[AllBs[[++i]]];
    Bstar=QEKStar[PosAss,B];
    If[SubsetQ[B,Aplus]&&Not[(SubsetQ[Bstar,Aplus])],
     Aplus=Union[Aplus,Bstar];Break[],flag=False]
    ];
   ];



  (* VERBOSITY *)
  If[OptionVerbosity>=10,Print["Calculating positive inferences."];];
  If[OptionVerbosity>=15,Print["Old NewPosAss: ",NewPosAss];];

  (* Adding new positive inferences to the old ones *)
  tmp=Union[NewPosAss,Map[({A,#})&,Aplus]];NewPosAss=tmp;

  (* VERBOSITY *)
```

```
    If[OptionVerbosity>=10,
     Print["New positive inferences:\n",NewPosAss];];


    (* Second, negative inferences: Calculating A-minus for a given A *)


    Aminus={};
    Do[
     B=QEKAssertionPremise[AllBs[[i]]];
     Bstar=QEKStar[PosAss,B];
     If[SubsetQ[A,Bstar],
      Aminus=Union[Aminus,SetDifference[Q,Bstar]]]
     ,{i,1,Length[AllBs]}];


    (* VERBOSITY *)
    If[OptionVerbosity>=10,Print["Calculating negative inferences."];];
    If[OptionVerbosity>=15, Print["Old NewNegAss: ",NewNegAss];];

    (* Add new negative inferences to the existing ones *)
    tmp=Union[NewNegAss,Map[({A,#})&,Aminus]];NewNegAss=tmp;

    (* VERBOSITY *)

    If[OptionVerbosity>=10,
     Print["New negative inferences:\n",NewNegAss];];


    (* Next A from the new block, or end of Scan[], i.e.  *)
    ],kBlockAs] ;


  (* Add NewPosAss, NewNegAss in PosAss and NegAss, and update OpenQ *)
  tmp=SetDifference[OpenQ,Union[NewPosAss,NewNegAss]];OpenQ=tmp;
  tmp=Union[NewPosAss,PosAss];PosAss=tmp;
  tmp=Union[NewNegAss,NegAss];NegAss=tmp;


  (* VERBOSITY *)
  If[OptionVerbosity>=10,
   Print["Open questions after incomplete inference rules:\n ",OpenQ];
   ];

  (* Find missing negative inferences of incomplete inference rules by

     finding contradictions in pseudoobservations within the new block *)
\


  ContradictionFound=True;
  While[ContradictionFound,
   ContradictionFound=False;
   Do[
    Aq=OpenQ[[i]];
    {VeryNewPosAss,VeryNewNegAss}=
     QEKCollectInferencesForPositiveAnswer[Q,PosAss,NegAss,Aq];
    (* Any contradiction? *)
```

```
      If[{Intersection[VeryNewPosAss,NegAss]!={}}||(Intersection[
            VeryNewPosAss,NegAss]!={}),
       ContradictionFound=True;

       (* VERBOSITY *)

       If[OptionVerbosity>=10,
        Print["Contradiction found for the assertion: ",Aq];
        ];

       (* Therefore, this Aq (and its inferences) are not open question,
        but rather an missed negative inference (therefore, the
            inferences are also missed) *)
       {VeryNewPosAss,VeryNewNegAss}=
        QEKCollectInferencesForNegativeAswer[Q,PosAss,NegAss,Aq];
       tmp=Union[VeryNewPosAss,PosAss];PosAss=tmp;
       tmp=Union[VeryNewNegAss,NegAss];NegAss=tmp;
       tmp=SetDifference[OpenQ,Union[VeryNewPosAss,VeryNewNegAss]];
       OpenQ=tmp;
       Break[];
       ];
     ,{i,1,Length[OpenQ]}];
   ];

  (* VERBOSITY *)
  If[OptionVerbosity>=10,
   Print["Open questions after missing inferences added:\n ",OpenQ];
   ];

  (* End of main While[] *)
  ];


(* Calculates structure obtained by assertions and returns it in
     appropriate format (assertions or space strucure)  *)

If[OptionOutputStructure===Assertions,
  tmp:={PosAss,NegAss,{}};,
    tmp=QEKOutput[Q,PosAss,NegAss,OutputStructure->SpaceStructure];
    ];

(* VERBOSITY *)
If[OptionVerbosity>=5,
  Print["Established structure by querying:\n ",tmp];];


(* SONDE *)
(* Write data to the querying session sonde  *)
If[OptionQSRReport==Report \[Or] OptionQSRReport==Both,
  QSR= QSRSondeQuestionsAskedReplace[QSR,NofQuestions];
  ];

(* VERBOSITY *) (* SONDE *)
If[OptionVerbosity>=10 &&
    (OptionQSRReport==Report\[Or] OptionQSRReport==Both) ,
  Print["Final querying session sonde status: ",QSR];
  ];

(* What to return? *)
Switch[OptionQSRReport,
```

```
    Result,Return[tmp],
    Report, Return[QSR],
    Both,Return[{tmp,QSR}]
    ];

  ] (* End of Module[] *)



(* QEOSQueryTextsInterpretations[NofSkils,
    TextPart] returns the TextPart (introduction, middle,
    ending) for the given number NofSkills (1,2, generic,
    all_minus_one)of skills *)
QEOSQueryTextsInterpretations[NofSkills_,TextPart0_]:=Module[
  {QueryTexts,TextParts,TextPart},
  TextParts={{introduction,2},{middle,3},{ending,4}};
  QueryTexts=
    {{1,"Suppose that a student under examination posses  the competence ",

      "Is possesing this skill (and none of the queried skills more) \
sufficient for solving the problem ",""},
     {2,
      "Suppose that a student under examination posses both the \
competences ",
      "Is possesing these skills (and none of the queried skills more) \
sufficient for solving the problem ",""},
     {generic,
      "Suppose that a student under examination posses all the skills: ",
      "Is possessing all of these skills (and none of the queried skills \
more) sufficient for solving the problem ",""},
     {all_minus_one,
      "Suppose that a student under examination  posses all of the \
queried skills except the skill ",
      "Is possesing these skills sufficient for solving the problem ",
      ""}};
  TextPart:=Select[TextParts,(#[[1]]===TextPart0)&,1][[1,2]];
  Return[Select[QueryTexts,(#[[1]]===NofSkills)&,1][[1,TextPart]]]
  ]

(* QEOSFormulateQueryInterpretations[a,Items,C,K,
    Skills]returns the query question on the given item a interpreted by \
Items and the competence state C of the competence structure K described \
through Skills *)
QEOSFormulateQueryInterpretations[a_,Items_,C_,K_,Skills_]:=Module[
  {NofSkills, nK,QueryTexts,QueryText,Text1,Text2,Text3,
    Text4,TextEnding,tmp,Ntmp},

  NofSkills=SetCardinality[C];
  (* Number of elementary competencies *)
  nK=SetCardinality[Apply[Union,K]];

  Which[
    NofSkills==1,
    Text1=QEOSQueryTextsInterpretations[NofSkills,introduction];
    Text2=SkillName[SkillGet[Skills,C[[1]]]];
    Text3=QEOSQueryTextsInterpretations[NofSkills,middle];
    Text4= ItemDescription[ItemGet[Items,a]];
    QueryText=StringJoin[{Text1,Text2,". ",Text3,Text4,"? "}];

    ,NofSkills==2,
```

```
    Text1=QEOSQueryTextsInterpretations[NofSkills,introduction];
    Text2=
     StringJoin[{SkillName[SkillGet[Skills,C[[1]]]]," and ",
        SkillName[SkillGet[Skills,C[[2]]]]}];
    Text3=QEOSQueryTextsInterpretations[NofSkills,middle];
    Text4= ItemDescription[ItemGet[Items,a]];
    QueryText=StringJoin[{Text1,Text2,". ",Text3,Text4,"? "}];

    (* all skills except one *)
    ,(nK-NofSkills)==1 ,
    Text1=QEOSQueryTextsInterpretations[all_minus_one,introduction];
    Text2=SkillName[SkillGet[Skills,SetDifference[Apply[Union,K],C][[1]]]];
    Text3=QEOSQueryTextsInterpretations[all_minus_one,middle];
    Text4= ItemDescription[ItemGet[Items,a]];
    QueryText=StringJoin[{Text1,Text2,". ",Text3,Text4,"? "}];

    (* generic *)
     ,True ,
    Text1=QEOSQueryTextsInterpretations[generic,introduction];
    (* construct the list of skill names given a competence state C *)
    tmp=Map[(SkillName[SkillGet[Skills,#]])&,C];
    Ntmp=Length[tmp];
    Text2=Apply[StringJoin,Table[tmp[[i]]<>", ",{i,1,Ntmp-1}]];
    Text2=Apply[StringJoin,{Text2,tmp[[Ntmp]]}];
    Text3=QEOSQueryTextsInterpretations[generic,middle];
    Text4= ItemDescription[ItemGet[Items,a]];
    QueryText=StringJoin[{Text1,Text2,". ",Text3,Text4,"? "}];
    ];

  TextEnding=
    " You may assume that chance factors, such as careless errors and lucky \
guesses, play no role in the student's performance.";

  Return[StringJoin[{QueryText,TextEnding}]];
  ]

(* QEOSQueryTextsSkills[PremiseSize,
    TextPart] returns the TextPart (introduction or question) for the \
premise of the size PremiseSize (1,2,generic,all_minus_one) *)
QEOSQueryTextsSkills[PremiseSize_,TextPart0_]:=
 Module[{QueryTexts,TextParts,TextPart},
  TextParts={{introduction,2},{question,3}};
  QueryTexts={{1,
      "Suppose that a student under examination does not posses the \
competence ",
      " Is it then practically certain that this student also does not \
posses the competence "},{2,
      "Suppose that a student under examination does not posses both the \
competencies ",
      " Is it then practically certain that this student also does not \
posses the competence "},{generic,
      "Suppose that a student under examination does not posses the \
following competencies: ",
      " Is it then practically certain that this student also does not \
posses the competence "},{all_minus_one,
      "Suppose you are tutoring a student who does not posses any of the \
competencies in the given set of competencies.",
      " Could you then start by teaching him the competence "}};
  TextPart:=Select[TextParts,(#[[1]]===TextPart0)&,1][[1,2]];
  Return[Select[QueryTexts,(#[[1]]===PremiseSize)&,1][[1,TextPart]]]
```

```
   ]

(* QEOSFormulateQuerySkills[E,Skills,
     Assertion] returns the query question on the given Assertion formulated \
for the human expert,
  given the set E of (elementary) skills and their interpretation Skills *)
QEOSFormulateQuerySkills[E_,Skills_,Assertion_]:=
  Module[{nE:=SetCardinality[E],
     nAssertion:=SetCardinality[QEKAssertionPremise[Assertion]],
     A:=QEKAssertionPremise[Assertion],b:=QEKAssertionConsequence[Assertion],
     QueryTexts,QueryText,Text1,Text2,Text3,Text4,tmp={},Ntmp},
   Which[
     nAssertion==1,
     Block[{},
       Text1:= QEOSQueryTextsSkills[1,introduction];
       Text2:=SkillName[SkillGet[Skills,A[[1]]]];
        tmp= QEOSQueryTextsSkills[1,question];
       Text3= Apply[StringJoin,{tmp,SkillName[SkillGet[Skills,b]],"?"}];
       QueryText:=StringJoin[{Text1,Text2,".",Text3}];
       ]
     ,nAssertion==2,
     Block[{},
       Text1= QEOSQueryTextsSkills[2,introduction];
       Text2=
         StringJoin[{SkillName[SkillGet[Skills,A[[1]]]]," and ",
            SkillName[SkillGet[Skills,A[[2]]]],"."}];
       tmp= QEOSQueryTextsSkills[2,question];
       Text3=Apply[StringJoin,{tmp,SkillName[SkillGet[Skills,b]],"?"}];
       QueryText:=StringJoin[{Text1,Text2,Text3}];
       ]
     ,(nE-nAssertion)==1,
     Block[{},
       Text1= QEOSQueryTextsSkills[all_minus_one,introduction];
       Text2="";
       tmp= QEOSQueryTextsSkills[all_minus_one,question];
       Text3=Apply[StringJoin,{tmp,SkillName[SkillGet[Skills,b]],"?"}];
       QueryText:=StringJoin[{Text1,Text2,Text3}];
       ]
     ,True (* GENERIC *),
     Block[{},
       Text1= QEOSQueryTextsSkills[generic,introduction];
       tmp=Map[(SkillName[SkillGet[Skills,#]])&,A];
       Ntmp=Length[tmp];
       Text2=Apply[StringJoin,Table[tmp[[i]]<>",",{i,1,Ntmp-1}]];
       Text2=Apply[StringJoin,{Text2,tmp[[Ntmp]],"."}];
       tmp= QEOSQueryTextsSkills[generic,question];
       Text3=Apply[StringJoin,{tmp,SkillName[SkillGet[Skills,b]],"?"}];
       QueryText:=StringJoin[{Text1,Text2,Text3}];
       ]
     ];
   Text4=
     " You may assume that chance factors, such as careless errors and lucky \
guesses, play no rule in the student's performance.";
   Return[StringJoin[{QueryText,Text4}]];
   ]

(* QEOSQueryExpertOnCompetenceSpace[E,SkOrExp,options] queryies a human
   (SkOrExp represents skills description) or a simulated expert (SkOrExp
      represents a simulated expert) is queried on the set E of elementary
   competencies and a simulated expert (SkOrExp represents the expert) and a
```

```
   structure (either space or assertions) is returned. The valid option is
   QueryingAlgorithm (Koppen [default]). For additional valid options
   see QEKQueryExpertByKoppen[]. *)

QEOSQueryExpertOnCompetenceSpace[E_,SkOrExp_, opts___]:=Module[
   {OptionQueryingAlgorithm,ReturnedValue},
   OptionQueryingAlgorithm=QueryingAlgorithm /. {opts} /.
      Options[QEOSQueryExpertOnCompetenceSpace];
   If[OptionQueryingAlgorithm==Koppen,
    (* QueryOn->CompetenceSpace unchangable *)
    ReturnedValue=QEKQueryExpertByKoppen[E,SkOrExp,
       QueryOn->CompetenceSpace,opts];
    ];
   Return[ReturnedValue]
   ]

(* MakeIAssertion[] *)
MakeIAssertion[CompetenceState_,Item_]:={Item,CompetenceState}

(* IAssertionItem[
    IAss] returns the item of the interpretation assertion IAss *)
IAssertionItem[IAss_]:=IAss[[1]]

(* IAssertionCompetenceState[
    IAss] returns the competence state of the interpretation assertion IAss \
*)
IAssertionCompetenceState[IAss_]:=IAss[[2]]

(* QEOSAskExpertOnInterpretation[K,Skills,Items,IAss,
    Human] returns an answer data structure on the interpretation assertion \
IAss from a human expert given a competence structure K with its description \
stored in the data file Skills and the interpretation assertion item desribed \
through Items *)
QEOSAskExpertOnInterpretation[K_,Skills_,Items_,IAss_,Human]:=Module[
   {answer,CertaintyFactor,a,C,E},
   E=Apply[Union,K];
   a=IAssertionItem[IAss];
   C=IAssertionCompetenceState[IAss];
   answer=Input[QEOSFormulateQueryInterpretations[a,Items,C,K,Skills]];
   CertaintyFactor=1;
   If[MemberQ[{"Y","T","YES","TRUE"},ToUpperCase[ToString[answer]]],
    answer=True,answer=False];
   Return[ExpertMakeAnswer[IAss,answer,CertaintyFactor]]
   ]

(* QEOSAskExpertOnInterpretation[IAss,Expert,
    Simulation] returns an answer data structure on the interpretation \
assertion IAss from the simulated Expert *)
QEOSAskExpertOnInterpretation[IAss_,Expert_,Simulation]:=Module[
   {answer,ExpCErrors,CertaintyFactor},
   answer=SetElementQ[IAssertionCompetenceState[IAss],
      CPAInterpretationOfItem[
       CPADiagnosticInterpretationFunction[ExpertMetaKnowledge[Expert]]
       ,IAssertionItem[IAss]]];
   ExpCErrors=ExpertCarelessErrors[Expert];
   CertaintyFactor=1;
   Return[ExpertMakeAnswer[IAss,answer,CertaintyFactor]]
   ]

(* QEOSInterpretationsOnItem[IASSes,
```

```
    q] returns all the interpretation assertions for the item q from the \
list IASSes  *)
QEOSInterpretationsOnItem[IASSes_,q_]:=Module[{},
  Select[IASSes,(IAssertionItem[#]===q)&]
  ]


(* QEOSInterpretationFunctionCollectInferencesForPositiveAnswer[OpenQ,
    IAss] returns all the interpretation assertion inferences given the \
list OpenQ
  of open interpretation assertions and interpretation assertion IAss
  that was judged positively *)
QEOSInterpretationFunctionCollectInferencesForPositiveAnswer[OpenQ0_,IAss_]:=


 Module[{OpenQ=OpenQ0,q=IAssertionItem[IAss]},
  OpenQ=QEOSInterpretationsOnItem[OpenQ,q];
  Select[
   OpenQ,((IAssertionItem[#]==IAssertionItem[IAss])&&
        SubsetQ[IAssertionCompetenceState[IAss],
         IAssertionCompetenceState[#]])&]
  ]


(* QEOSInterpretationFunctionCollectInferencesForNegativeAnswer[OpenQ,
    IAss] returns all the interpretation assertion inferences given the \
list OpenQ
  of open interpretation assertions and interpretation assertion IAss
  that was judged negatively *)
QEOSInterpretationFunctionCollectInferencesForNegativeAnswer[OpenQ0_,IAss_]:=


 Module[{OpenQ=OpenQ0,q=IAssertionItem[IAss]},
  OpenQ=QEOSInterpretationsOnItem[OpenQ,q];
  Select[
   OpenQ,((IAssertionItem[#]==IAssertionItem[IAss])&&
        SubsetQ[IAssertionCompetenceState[#],
         IAssertionCompetenceState[IAss]])&]
  ]

(* QEOSInterpretationFunctionCountInferences[IASSes,
    IAss] returns the data structure {Neg,
    Pos} containing the number Neg of inferences when negative answer on \
the interpretation assertion IAss was obtained (given the list IASSes of open \
interpretation assertions),
 and the number Pos of inferences in case of positive answer. *)
QEOSInterpretationFunctionCountInferences[IASSes_,IAss_]:=Return[
  {Length[
     QEOSInterpretationFunctionCollectInferencesForNegativeAnswer[IASSes,
      IAss]],Length[
     QEOSInterpretationFunctionCollectInferencesForPositiveAnswer[IASSes,
      IAss]]}
  ]

(* QEOSInterpretationFunctionSelectionRuleMaximin[
    OpenQ] returns an interpretation assertion which has maximal exected \
number of inferences according to the maximin selection rule  *)
QEOSInterpretationFunctionSelectionRuleMaximin[OpenQ_]:=Module[
  {candidates,MaxIAss,i},
  candidates=Map[({#[[1]],Min[#[[2,1]],#[[2,2]]]})&,
```

```
      Map[({#,QEOSInterpretationFunctionCountInferences[OpenQ,#]})&,
       OpenQ]];
   MaxIAss=candidates[[1]];
   Do[
    If[candidates[[i,2]]>MaxIAss[[2]],MaxIAss=candidates[[i]]];
    ,{i,2,Length[candidates]}];
   Return[MaxIAss[[1]]]
   ]

(* QEOSInterpretationFunctionSelectionRuleWeightedSums[
     OpenQ] returns an interpretation assertion which has maximal exected \
number of inferences according to the weighted sums selection rule  *)

QEOSInterpretationFunctionSelectionRuleWeightedSums[OpenQ_,opts___]:=
  Module[{PrYES,PrNO,candidates,MaxIAss,i},
   PrYES=
    ProbabilityYES /. {opts} /.
     Options[QEOSInterpretationFunctionSelectionRuleWeightedSums];
   PrNO=1-PrYES;
   candidates=Map[({#[[1]],(PrNO*#[[2,1]]+PrYES*#[[2,2]])})&,
      Map[({#,QEOSInterpretationFunctionCountInferences[OpenQ,#]})&,
       OpenQ]];
   MaxIAss=candidates[[1]];
   Do[
    If[candidates[[i,2]]>MaxIAss[[2]],MaxIAss=candidates[[i]]];
    ,{i,2,Length[candidates]}];
   Return[MaxIAss[[1]]]
   ]

(* QEOSSelectNextInterpretationQuestion[OpenQ,
     opts] returns the next interpretation assertion question given the \
(sorted) list OpenQ of open interpretation assertions respecting the given \
options. Possible options are:
     OpenQuestions (Sorted [default],Unsorted) -
       in case the list OpenQ is not sorted,
  sorts it according to the cardinality of competence states;
  RuleScope (Block [default], All) -
   whether to search for the next question only in the next block (defined \
by the cardinality of competence states) or searching all the interpretation \
assertions in OpenQ; SelectionRule (Maximin [defult], WeightedSums).  *)


QEOSSelectNextInterpretationQuestion[OpenQ0_,opts___]:=
  Module[{OptionSelectionRule,OptionRuleScope,OptionOpenQuestions,tmp,
     min,OpenQ=OpenQ0,NextIAss},

   If[OpenQ==={},Return[{}]];


   (* if the list of open questions is unsorted, sort it according to the
      size of interpretation assertion competence states *)

   OptionOpenQuestions=OpenQuestions /. {opts} /.
      Options[QEOSSelectNextInterpretationQuestion];
   If[¬(OptionOpenQuestions===Sorted),
    tmp=Sort[OpenQ,(Length[#1[[2]]]<Length[#2[[2]]])&];OpenQ=tmp;
    ];


   (* If the scope of the selection rule is Block, extract the block of
```

```
      interpretation assertions with the minimal competence states *)
   OptionRuleScope=RuleScope /. {opts} /.
     Options[QEOSSelectNextInterpretationQuestion];
   If[(OptionRuleScope===Block),
    min=Length[OpenQ[[1,2]]];
    tmp=Select[OpenQ,(Length[#[[2]]]===min)&];OpenQ=tmp;
    ];


   (* Select the next interpretation assertion question *)
   OptionSelectionRule=SelectionRule /. {opts} /.
     Options[QEOSSelectNextInterpretationQuestion];
   Switch[OptionSelectionRule
    ,WeightedSums,
    NextIAss=QEOSInterpretationFunctionSelectionRuleWeightedSums[OpenQ];
    ,Maximin,
    NextIAss=QEOSInterpretationFunctionSelectionRuleMaximin[OpenQ];
    ];
   Return[NextIAss]
   ]

(* QEOSQueryExpertOnInterpretationFunction[EK,A,DescOrExp,opts] establish an
     interpretation function by querying an expert given the set A
     of items and the competence space EK.
       After the querying only the competence-based items set,
  together with its interpretations, is kept. In case of human expert,
  DescOrExp
    represents structure {Skills,Items},
  where the used skills are described through Skills and the items through \
Items. In case of simulated expert,  DescOrExp
    describes the expert. The possible options are:
     Verbosity (Off [Default],
        1,5,10,15), QSRReport (Result [default], Report or Both),
  Expert (Human [default], Simulation), SelectionRule (Maximin
      [default], WeightedSums) and RuleScope (Block [default], All).  *)

QEOSQueryExpertOnInterpretationFunction[EK_,A_,DescOrExp_,opts___]:= Module[
   {K,OpenQ,PosIAss,NegIAss,Answer,Skills,Items,SimExpert,tmp,PosAnsInf,
    NegAnsInf,k={},QSR, OptionRuleScope,OptionSelectionRule,
    OptionVerbosity,
    OptionHumOrSim,OptionQSRReport},

   (* VERBOSITY LEVEL *)
   OptionVerbosity=Verbosity /.
       {opts} /. Options[QEOSQueryExpertOnInterpretationFunction];

   (* What the function should return as its output? *)
   OptionQSRReport= QSRReport /. {opts} /.
     Options[QEOSQueryExpertOnInterpretationFunction];


   (* Whether to query human or simulated expert?*)
   OptionHumOrSim=Expert /.
       {opts} /. Options[QEOSQueryExpertOnInterpretationFunction];

   (* The rule used for selecting the next question *)
   OptionSelectionRule=SelectionRule /. {opts} /.
     Options[QEOSQueryExpertOnInterpretationFunction];

   (* Scope of the selection rulu while considering candidates *)
```

```
OptionRuleScope=RuleScope /.{opts} /.
   Options[QEOSQueryExpertOnInterpretationFunction];

(* VERBOSITY *)
If[OptionVerbosity>=1,
 Print["QEOS v1.0, Querying Session Report.\n",
   "A Straightforward procedure for querying an expert (",OptionHumOrSim,
   ") on interpretation function.\n"
   ,OptionSelectionRule," selection rule used with the ",
   OptionRuleScope,
   " scope."];
 Print["Function returns (Result, QSRReport or Both):",OptionQSRReport];
 ];



(* Parameter DescOrExp means represents either an expert or a pair {S,I}
   where S stands for skills description, and I for items description *)


If[OptionHumOrSim==Human,
 Skills=DescOrExp[[1]];
 Items=DescOrExp[[2]];
 ];
If[OptionHumOrSim==Simulation,
 SimExpert=DescOrExp;
 ];


(* VERBOSITY *)
If[OptionHumOrSim==Human && OptionVerbosity>=5,
 Print["Description of used skills:\n",Skills];
 Print["Description of used items: \n",Items];
 ];
If[(OptionHumOrSim==Simulation) && (OptionVerbosity>=5),
 Print["Simulated expert:\n ",SimExpert];
 ];


(* Creating the list of all open questions and sorting them
   according to their premise sizes *)
K=StructureFamily[EK];
OpenQ=SetCartesianProduct[A,SetDifference[K,{{}}]];
tmp=Sort[OpenQ,(Length[#1[[2]]]<Length[#2[[2]]])&];OpenQ=tmp;
(* TO EXCLUDE THE WHOLE SET KAPA ALSO? *)

(* Initializing positive and negative interpretation assertions *)
PosIAss=NegIAss={};

(* Count number of questions asked *)
NofQuestions=0;


(* SONDE *)
(* Create and initialize a querying session sonde if requested *)
If[OptionQSRReport==Report || OptionQSRReport==Both,
 QSR=QSRMakeSonde[0,0];
 QSR=QSRSondeOpenQuestionsInitialReplace[QSR,Length[OpenQ]];
 QSR= QSRSondeQuestionsAskedReplace[QSR,NofQuestions];
 ];
```

```
(* VERBOSITY *) (* SONDE *)
If[OptionVerbosity>=10 &&
   (OptionQSRReport==Report|| OptionQSRReport==Both) ,
  Print["Initial querying session sonde status: ",QSR];
  ];




(* Any interpretation assertion left open? *)
While[OpenQ!={},


  (* VERBOSITY *)
  If[OptionVerbosity>=5,
    Print["Open questions:\n",OpenQ];
    Print["Known positively judged interpretation assertions:\n",
      PosIAss];
    ];


  (* Select the next open interpretation assertion for querying *)
  NextIAss=
    QEOSSelectNextInterpretationQuestion[OpenQ,OpenQuestions->Sorted
      ,RuleScope->OptionRuleScope,SelectionRule->OptionSelectionRule];


  (* Query a human or simulated expert for an answer *)
  If[OptionHumOrSim==Human,
    Answer=QEOSAskExpertOnInterpretation[K,Skills,Items,NextIAss,Human];
    NofQuestions++;
    ];
  If[OptionHumOrSim==Simulation,
    Answer=QEOSAskExpertOnInterpretation[NextIAss,SimExpert,Simulation];
    NofQuestions++;
    ];



  (* VERBOSITY *)
  If[OptionVerbosity>=5,
    Print[NofQuestions,". question: ",NextIAss,
      ". The expert's answer: ",Answer];
    ];


  (* Draw the inferences *)
  PosAnsInf=NegAnsInf={};

  (* Add both the answer and inferences, update the list
     of judged interpretation assertions *)
  If[ExpertAnswerLogicalValue[Answer],
    PosAnsInf=
      QEOSInterpretationFunctionCollectInferencesForPositiveAnswer[OpenQ,
        ExpertAnswerAssertion[Answer]];
    tmp=Join[PosIAss,PosAnsInf];PosIAss=tmp;
    ,NegAnsInf=
      QEOSInterpretationFunctionCollectInferencesForNegativeAnswer[OpenQ,
```

```
      ExpertAnswerAssertion[Answer]];
   ];

  (* Update the list of open questions *)
  tmp=SetDifference[OpenQ,Union[NegAnsInf,PosAnsInf]];OpenQ=tmp;

  (* VERBOSITY *)
  If[OptionVerbosity>=5,
   Print["The answer and its inferences added to the known",
    " interpretation assertions."];
   ];
  (* VERBOSITY *)
  If[OptionVerbosity>=10,
   Print["Positive answer inferences: \n ",PosAnsInf];
   Print["Negative answer inferences: \n ",NegAnsInf];
   ];


  (* VERBOSITY *)
  If[OptionVerbosity>=5,
   Print["New positively judged interpretation assertions:\n",
    PosAnsInf];
   Print["Open questions left:\n ",OpenQ];
   ];


  ]; (* End of While[] *)

(* Form the established interpretation function *)
k=CPAInterpretationFunctionUpgrade[
   CPAInterpretationFunctionInitialize[A,K],
   QEOSInterpretationsFromIAssertions[PosIAss]];

(* VERBOSITY *)
If[OptionVerbosity>=10,
 Print["Established interpretations by querying:\n",FunctionMapping[k]];
 ];


(* Keep only the interpretations for competence-space based items, and
   return the interpretation function *)
tmp=CPAInterpretationFunctionConformTo[k];

(* VERBOSITY *)
If[OptionVerbosity>=10,
 Print["Established interpretations of competency-space",
  " based items:\n",FunctionMapping[tmp]];
 ];


(* SONDE *)
(* Write data to the querying session sonde  *)
If[OptionQSRReport==Report || OptionQSRReport==Both,
 QSR= QSRSondeQuestionsAskedReplace[QSR,NofQuestions];
 ];

(* VERBOSITY *) (* SONDE *)
If[OptionVerbosity>=10 &&
   (OptionQSRReport==Report|| OptionQSRReport==Both) ,
 Print["Final querying session sonde status: ",QSR];
```

```
    ];

  If[OptionQSRReport==Both,Return[{tmp,QSR}]];
  If[OptionQSRReport==Report,Return[QSR]];
  If[OptionQSRReport==Result,Return[tmp]];

  ] (* End of Module[] *)

(* QEOSInterpretationsFromIAssertions[
    IASSes] returns the list of (partial) interpretations formed from the \
list of interpretation assertions *)
QEOSInterpretationsFromIAssertions[IASSes_]:=Map[({#[[1]],{#[[2]]}})&,IASSes]

(*
 QEOSQueryExpertOnDiagnostic[E,A,DescOrExp,
      opts] queryies a human expert (DescOrExp equals to {Skills,
        Items} describes the skills and the items) or a simulated expert \
(DescOrExp represents a simulated expert whose
        meta-knowledge must be given as a diagnostic) on competence-
  performance
    diagnostic given the set A of items and the set E of elementary
    competencies. The valid options are: Expert (Human [default] or
        Simulation),  QSRReport (Result [default], Report or
       Both). For other valid options see:
    QEOSQueryExpertOnCompetenceSpace[]
     and  QEOSQueryExpertOnInterpretationFunction[]. *)

QEOSQueryExpertOnDiagnostic[E_,A_,DescOrExp_,opts___]:=Module[
  {qsr,qsr1,qsr2,Exp,Skills,Items,K,k,DIAG,OptionHumOrSim},

  (* Whether to query human or simulated expert *)
  OptionHumOrSim:=Expert /. {opts}/. Options[QEOSQueryExpertOnDiagnostic];

  (* What the function should return as its output? *)
  OptionQSRReport:=QSRReport /. {opts}/.
     Options[QEOSQueryExpertOnDiagnostic];


  (* VERBOSITY *)
  If[OptionVerbosity>=1,
   Print["Function returns (Result, Report or Both): ",OptionQSRReport];
   ];


  (* Query on competence space *)
  (* In case of querying a human expert for competence space,
   pass only skills description *)
  If[OptionHumOrSim==Human,
   {K,qsr1}=QEOSQueryExpertOnCompetenceSpace[E,DescOrExp[[1]],
     QSRReport->Both,opts];
   ];
  If[OptionHumOrSim==Simulation,
   {K,qsr1}=
    QEOSQueryExpertOnCompetenceSpace[E,DescOrExp,Expert->Simulation,
     QSRReport->Both,opts];
   ];


  (* Query on interpretation function *)
  {k,qsr2}=QEOSQueryExpertOnInterpretationFunction[K,A,DescOrExp,
```

```
      QSRReport->Both,opts];


   (* Calculate the diagnostic *)
   DIAG=CPADiagnosticFromInterpretationFunction[k];


   (* SONDE *) (* Combine two observations *)
   qsr=qsr1+qsr2;

   (* What to return? *)
   Switch[OptionQSRReport,
     Result,Return[DIAG],
     Report,Return[qsr],
     Both,Return[{DIAG,qsr}]
     ];

   ] (* End of Module[] *)



(* QSRMakeSonde[OQB,NQ] returns the querying session (QSR) sonde consisting
   of the number OQ of open questions at the beginning of the observing
    and the number NQ of question given to an expert during
   the observation. *)
QSRMakeSonde[OQ_Integer,NQ_Integer]:={OQ,NQ}

(* QSRSondeOpenQuestionsInitial[Sonde] returns the initial number of open
   questions as recorded by the querying session Sonde*)
QSRSondeOpenQuestionsInitial[sonde_]:=sonde[[1]];

(* QSRSondeQuestionsAsked[Sonde] returns the number of questions asked as
   recorded by the querying session Sonde *)
QSRSondeQuestionsAsked[sonde_]:=sonde[[2]];

(* QSRSondeOpenQuestionsInitialReplace[Sonde,N] returns the querying
   session Sonde with the initial number of open questions replaced with N*)


QSRSondeOpenQuestionsInitialReplace[Sonde_,N_]:=ReplacePart[Sonde,N,1]

(* QSRSondeOpenQuestionsInitialAdd[Sonde,
     N] returns the querying session Sonde with the initial number of open \
questions incremented by N *)
QSRSondeOpenQuestionsInitialAdd[Sonde_,N_]:=ReplacePart[Sonde,
   QSRSondeOpenQuestionsInitial[Sonde]+N,1]

(* QSRSondeQuestionsAskedReplace[Sonde,N] returns the querying session
   Sonde with the number of questions asked replaced with N *)
QSRSondeQuestionsAskedReplace[Sonde_,N_]:=ReplacePart[Sonde,N,2]

(* QSRSondeQuestionsAskedAdd[Sonde,N] returns the querying session
   Sonde with the number of questions asked incremented by N*)
QSRSondeQuestionsAskedAdd[Sonde_,N_]:=ReplacePart[Sonde,
   QSRSondeQuestionsAsked[Sonde]+N,2]



(* QEQueryExpertOnKnowledgeSpace[Q,ItOrExp,options] queryies a human
```

```
   (ItOrExp represents items description) or a simulated expert (ItOrExp
      represents a simulated expert) on the set Q of items , and a
   structure (either space or assertions) is returned. The valid option is
   QueryingAlgorithm (Koppen [default]). For additional valid options
   see QEKQueryExpertByKoppen[]. *)

QEQueryExpertOnKnowledgeSpace[Q_,ItOrExp_,opts___]:=Module[
   {OptionQueryingAlgorithm,ReturnedValue},
   OptionQueryingAlgorithm=QueryingAlgorithm /. {opts} /.
      Options[QEQueryExpertOnKnowledgeSpace];
   If[OptionQueryingAlgorithm==Koppen,
    (* QueryOn->KnowledgeSpace unchangable *)
    ReturnedValue=QEKQueryExpertByKoppen[Q,ItOrExp,
       QueryOn->KnowledgeSpace,opts];
    ];
   Return[ReturnedValue]
   ]




End[]



EndPackage[]
```

KSMP assessment module

```
(* :Title: KnowledgeSpaces Mathematica Package, Assessment Module *)

(* :Context: KnowledgeSpaces`Assessment` *)

(* :Author: Andrej Zaluski *)

(* :Summary:
        Deterministic Knowledge Assessment.   *)

(*  :Copyright: 20001, Andrej Zaluski.
        See attached license.
 *)

(*  :Package Version: 0.91 beta 1 *)

(*  :Mathematica Version: 4.0 *)

(*  :History:
          1.0 the first published version.
          0.91 beta 1
 *)

(*  :Keywords:  knowledge space theory; knowledge assessment *)

(*  :Sources: *)

(*  :Warnings: BETA 1 - TESTING PHASE! *)

(*  :Limitations: *)

(*  :Discussion: *)

(*  :Requirements:
                  KnowledgeSpaces`L2` *)




BeginPackage["KnowledgeSpaces`Assessment`","KnowledgeSpaces`L1`","KnowledgeSpaces`L2`"];


KAAskStudent::usage="KAAskStudent[Stud,QE,\"Simulation\"] returns an answer \
from the simulated student Stud on the question QE.\n\nKAAskStudent[Items,QE,\
\"Human\"] returns an answer from a human student on the question QE \
described in Items.";

KADSelectNextQuestion::usage="KADSelectNextQuestion[SF,{CA,NCA}]returns the \
next item to be asked by the deterministic knowledge assessment procedure on \
the knowledge structure SF assuming already collected correct CA and \
incorrect NCA answers by a student. \n\nKADSelectNextQuestion[SF] assumes \
that no previous answers have been collected.";

KADAssessKnowledge::usage="KADAssessKnowledge[SF,ItemsStudent,options]\
performs the deterministic assessment of a student's knowledge given the \
knowledge structure SF. In case of a human student (option Examinee, value \
\"Human\" [default]) the parameter ItemsStudent takes the description of \
```

```
items, while in case of a simulated student (option Examinee, value \
\"Simulation\") it stores a simulated student. The function may return either \
a knowledge state (option AssessmentResult, value \"State\" [default]) or an \
assessment history (option AssessmentResult, value \"History\"). The \
additional valid option is Verbosity (Off [default],1,5,10,20).";

Examinee::usage="Examinee is an option of KADAssessKnowledge. The admissible \
values are \"Human\" and \"Simulation\". ";

AssessmentResult::usage="AssessmentResult is an option of KADAssessKnowledge. \
The admissible values are \"State\" and \"History\".";

Options[KADAssessKnowledge]={Examinee->"Human",AssessmentResult->"State",
   Verbosity:>$Verbosity};



Begin["`Private`"]



KAFormulateQuestion::usage="KAFormulateQuestion[Items,QE]returns the \
question based on the item QE described by Items.";

KAFormulateQuestion[Items_,q_]:=Module[
   {item,text,t2="?\n\nThe possible answers are:\n\n"},
   item=ItemGet[Items,q];
   text=ItemText[item]<>t2<>
      StringJoin[
       Map[(ToString[#[[1]]]<>".  "<>#[[2]]<>".\n")&,ItemAnswers[item]]];
   Return[text];
   ]

KADSelectNextQuestionAuxFirst[Structure[Q0_List?SetNonEmptyQ,K0_]]:=
         Module[{Q=Q0,K=K0,QK=Structure[Q0,K0],maxq,balans,i},
           balans=Map[({#1,Abs[SetCardinality[FX[QK,List[#1],{}]]-
                       SetCardinality[Complement[K,FX[QK,List[#1],{}]]]]})&,
                Q] ;
    maxq=First[balans] ;
    Do[
          If[maxq[[2]]>balans[[i,2]],
               maxq=balans[[i]] ]
                  ,{i,Length[balans]}  ]  ;
    Return[maxq[[1]]]
          ]

KADSelectNextQuestionAuxNonFirst[Structure[Q0_,K0_],CAnswers_,NCAnswers_]:=
 Module[
   {QK=Structure[Q0,K0],Q=Q0,States,q={}},
   States=FX[QK,CAnswers,NCAnswers];
   If[SetCardinality[States]>1,
    q=KADSelectNextQuestionAuxFirst[
       Structure[DeleteCases[Q,Union[CAnswers,NCAnswers]],
         States]]
    ];
   Return[q]
   ]


KAAskStudent[Stud_,q_,"Simulation"]:=Module[{CF=1,LV3:=False,tmp},
```

```
   If[SubsetQ[{q},StudentKnowledge[Stud]],LV3:=True];
   tmp:=StudentMakeAnswer[q,LV3,CF];
   Return[tmp]
   ]
KAAskStudent[Items_,q_,"Human"]:=Module[{answer,CF=1,LV},
   answer=Input[KAFormulateQuestion[Items,q]];
   If[answer==ItemCorrectAnswer[ItemGet[Items,q]],
    LV=True,LV=False
    ];
   Return[StudentMakeAnswer[q,LV,CF]]
   ]

KADSelectNextQuestion[Structure[S_,F_]]:=
 Module[{SF=Structure[S,F]},
   KADSelectNextQuestion[SF,{{},{}}]]

KADSelectNextQuestion[Structure[S_,F_],CNC_List]:=
 Module[{SF=Structure[S,F],q,C=CNC[[1]],NC=CNC[[2]]},
   If[Union[C,NC]=={},
    q:=KADSelectNextQuestionAuxFirst[SF];,
    q:=KADSelectNextQuestionAuxNonFirst[SF,C,NC];
    ];
   Return[q];
   ]

KADAssessKnowledge[Structure[Q_,K_],ItemsStudent_,opts___]:=
Module[{OpenQ,OpenK,q,Qy={},Qn={},Items,Student,Hist={},
    OptionAssessmentResult,OptionExaminee,OptionVerbosity,ans,tmp},

   OptionVerbosity=Verbosity /.{opts}/.Options[KADAssessKnowledge];

   OptionExaminee= Examinee /. {opts} /. Options[KADAssessKnowledge];

   OptionAssessmentResult=
    AssessmentResult /. {opts} /. Options[KADAssessKnowledge];


   (* VERBOSITY *)
   If[OptionVerbosity>=1,
    Print[
     "KSMP v1.0, Deterministic Knowledge Assessment Report.\nDeterministic \
assessment procedure used on a student (",OptionExaminee,
      ") and the result returned as  ",
      OptionAssessmentResult, "."];
    ];

   OpenQ=Q;OpenK=K;
   While[SetCardinality[OpenK]>1,
    q=KADSelectNextQuestion[Structure[Q,K],{Qy,Qn}];

    (* VERBOSITY *)
    If[OptionVerbosity>=10,
     Print["Selected item: ", q];
     ];

    Switch[OptionExaminee,"Simulation",
     ans=KAAskStudent[ItemsStudent,q,"Simulation"];,
     "Human",ans=KAAskStudent[ItemsStudent,q,"Human"];
     ];
    AppendTo[Hist,ans];
```

```
    If[StudentAnswerLogicalValue[ans],
      AppendTo[Qy,StudentAnswerItem[ans]];,
      AppendTo[Qn,StudentAnswerItem[ans]];
      ];

    (* VERBOSITY *)
    If[OptionVerbosity>=5,
      Print["The answer: ",ans,"\nCollected positive answers: ",Qy,
        "\nCollected negative answers: ",Qn];
      ];

    OpenK=FX[Structure[Q,K],Qy,Qn];

    (* VERBOSITY *)
    If[OptionVerbosity>=10,
      Print["Remaining states:", OpenK];
      ];


    ];
   Switch[OptionAssessmentResult,
    "State",tmp=Qy;,
    "History",tmp=Hist;
    ];
   Return[tmp]
   ]

End[ ]

EndPackage[]
```